CANDIDE, AN INTERACTIVE SYSTEM FOR THE ACQUISITION OF DOMAIN SPECIFIC KNOWLEDGE

Final Report
Covering the Period September 7, 1984 to January 31, 1988

May 1988

By: Fernando Pereira, Senior Computer Scientist

Prepared for:

Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, Virginia 22209

Attention: Commander Allen Sears

ARPA Order No. 3988

Contract N00039-84-C-0109

SRI Project 7783

Preparation of this paper was supported by the Department of the Navy under Contract N00039-84-C-0109 with the Space and Naval Warfare Systems Command.

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025-3493
(415) 326-6200
TWX: 910-373-2046
Telex: 334486

SRI International®

Contents

Chapter 1

Introduction

Over the past several decades, computer systems have come to embody more and more complex forms of knowledge. For example, a production-line control system must embody knowledge about how the steps in the production line are ordered, what the capacity of each machine in the system is, and what the procedure is for assigning high priority to certain "hot" lots. As another example, consider a system that monitors for malfunctions in a space vehicle and suggests corrective actions. It must embody knowledge about the normal configuration of the vehicle's components, about the symptoms of component malfunction—for instance, that a sudden pressure rise may indicate that a valve is jammed - and about the steps that should be taken to correct any malfunction. Even systems that are more like traditional databases may need to embody relatively complex knowledge. A useful personnel database system must not only keep a consistent and complete record of who works where in an organization, but must also recognize that each employee has a single boss who is also an employee, and that while by law all employees must earn more than $3.50 per hour, in fact no one in the company earns less than $4.15 per hour, and so on.

How is the knowledge that is embodied in a computer system acquired? In most present-day systems, it is painstakingly encoded in the actual algorithms that implement the system. Even in most artificial intelligence systems, which may have a level of explict representation of the knowledge, this knowledge must be entered into the system as expressions in some formal computer language. Usually this must be done by a specialized programmer or knowledge engineer. The result is a *knowledge bottleneck*. Significant efficiency could be gained by making it possible for the needed knowledge to be installed by those who are knowledgeable about the domain of some computer system, but who do not necessarily possess expertise about the internal workings or language of the system itself.

The Candide project, whose result are described in this report,has been concerned with developing tools that will help to alleviate the knowledge bottleneck problem. Under its auspices, the Candide system was designed and implemented. Candide is a multimodal interpretation system designed specifically for knowledge acquisition tasks. Candide makes it possible for someone who is knowledgeable about the domain of some computer system to create, add to, or update the system's knowledge base using a combination of ordinary English discourse and a simple graphical interface tool.

The current version of Candide uses the Procedural Reasoning System (PRS) [10] as a testbed. PRS is a reactive, real-time system for reasoning about and performing tasks in

dynamic environments. An important part of PRS's knowledge base is its set of procedural networks, each of which encodes information about the steps of some domain procedure and which can by used by PRS for performing the procedure. One important application of PRS has been to the problem of equipment malfunction on NASA's space shuttle[1]. Candide allows an engineer familiar with the procedures for dealing with space-shuttle equipment malfunction to build and update the collection of procedural networks that PRS needs for this application.

Significantly, Candide is not just a single-utterance interpretation system; it includes capabilities for processing the types of extended natural-language dialogues that are necessary in complex tasks such as knowledge acquisition. Major contributions of the Candide project include the development of a unified framework for semantic and pragmatic processing of natural-language discourse, supported by powerful and general syntactic parsing mechanisms. A number of significant advances in unification-based parsing have been made during the course of this project. In addition, techniques have been developed for reasoning about the domain and discourse history in order to handle a wide range of important semantic and pragmatic phenomena, including definite and indefinite reference, anaphora, quantifier scoping, resolution of nominal compounds, resolutions of syntactic and lexical ambiguity, and metonomy.

In this report, we describe the results of the basic research program done on the Candide project. The present chapter contains a brief overview of the main areas covered in the research and a guide to the material presented in later chapters. Each chapter that follow is a stand-alone document that examines some aspect of the Candide concept. Appendices A, B, and C present users' manuals various parts of the system.

1.1 The Acquisition of Procedural Knowledge

As noted above, the current version of Candide is used to build and update the collection of procedural networks that makes up a crucial part of PRS's knowledge base. Each procedural network encodes information about the steps of some domain procedure which can then be used by PRS for performing that procedure.

Details of the ways in which PRS uses the networks can be found in [1,10]. Here we simply provide enough information to make the operation of the Candide system clear. Every procedural network comprises a set of invocation conditions that specify when the encoded procedure is relevant, and a graph that encodes the steps of the procedure itself. Arcs of the graph are labeled in one of three ways: there are achievement arcs, which are prefixed with the operator "!"; query arcs, prefixed with the operator "?"; and assertional arcs, prefixed with the operator "→". Each arc type has an associated method of traversal: for example, an query arc can be traversed only if the condition on it is true. What is important to note for our purposes is that the three arc types correspond quite directly to the three major moods of English sentences.

Figure 1.1 depicts a portion of a highly simplified procedural net. It has been adapted from one application of PRS: monitoring for and responding to equipment malfunctions on NASA's space shuttle. The figure shows a small portion of a network that encodes the procedure for dealing with the failure of one of the jets in the shuttle's reaction control system (RCS). The topmost arc corresponds to an imperative, "Close the manifold."; the

two arcs emanating from node 1 correspond to an interrogative, "Is there high usage in the RCS?"; and the arc emanating from arc 2 corresponds to an assertion, "The jet driver has an electrical failure." Invocation conditions either may be assertions, which denote propositions that, when true, indicate that the procedure is relevant, or they may be imperatives, which denote goals that are likely to be satisfied as a result of performing the procedure.

Using Candide, an engineer familiar with the procedures relevant to some domain can readily create a procedural network by first describing the invocation conditions for the procedure, and subsequently, by constructing an arc of the network, specifying in English the conditions on that arc, and then repeating the process as necessary. The engineer thus specifies complex objects, conditions, goals, actions, and propositions in English, while their temporal and causal relations to one another are specified graphically. Candide automatically translates each English condition into a statement in the language used by PRS. The translated condition is displayed on the network; the English statement is also stored for the convenience of the user. The construction of arcs and nodes is achieved by the graphical interface tool Grasper [3]. Appendix A of this document is a User's Manual for building PRS networks using Candide.

1.2 The Architecture of Candide

The Candide system is implemented in Prolog and Zeta-Lisp on Symbolics 3600-series computers. Figure 1.2 provides a schematic diagram of the system. Processes are shown in rectangles, and data stores in ovals. Candide has been embedded in Grasper, which itself can be called directly from PRS.

The Candide system can be decomposed into three major components: a syntactic parser (PATR-II), a system for semantic and pragmatic interpretation (CANDIDE-SPI), and a routine that transforms the logical forms produced by CANDIDE-SPI into expressions in PRS's internal language (TRANSFORM). The latter two components have been kept distinct in an effort to maintain Candide's generality and portability. The logical forms produced by CANDIDE-SPI are general enough that they can subsequently be translated into a variety of formal languages that might be used by different end systems. Work completed on the two large components (PATR-II and CANDIDE-SPI) is described further below.

1.3 Syntactic Analysis

Syntactic analysis in Candide is performed by PATR-II, a state-of-the-art syntactic parsing system that implements unification-based parsing. The development of PATR-II was begun prior to the start of the Candide project, under the auspices of DARPA Contract No. N00038-84-K-0078: *Research on Interactive Acquisition and Use of Knowledge*, also known as the KLAUS project. When funding on KLAUS was terminated, some of the work on PATR was incorporated into the Candide project. Because termination of funding resulted in there not being a final report produced for the KLAUS project, the work that was incorporated into Candide is reported on here. In addition, a good deal of work on the PATR-II system was performed directly under the support of the Candide project, and a

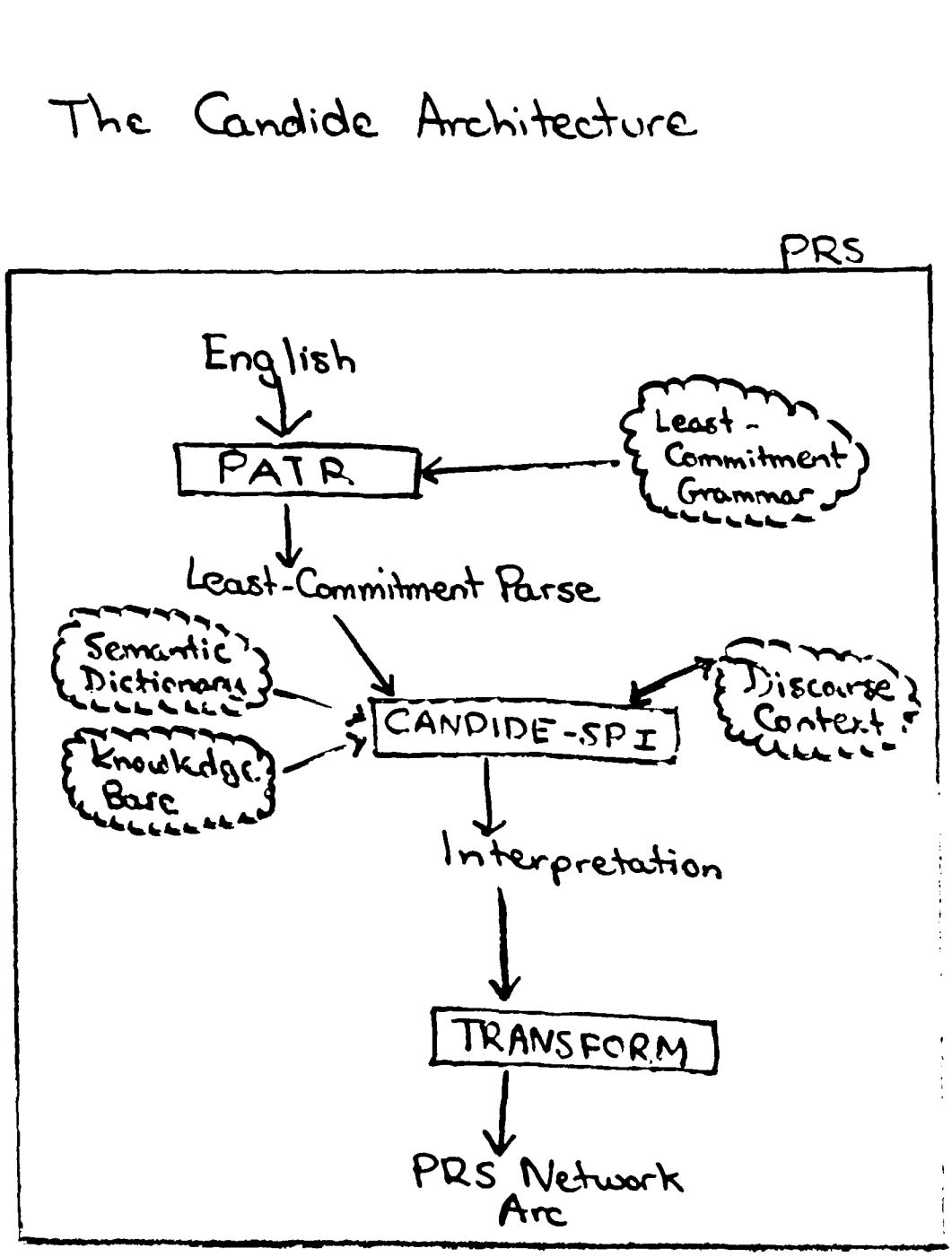


Figure 1.1: A Simplified PRS Procedural Net

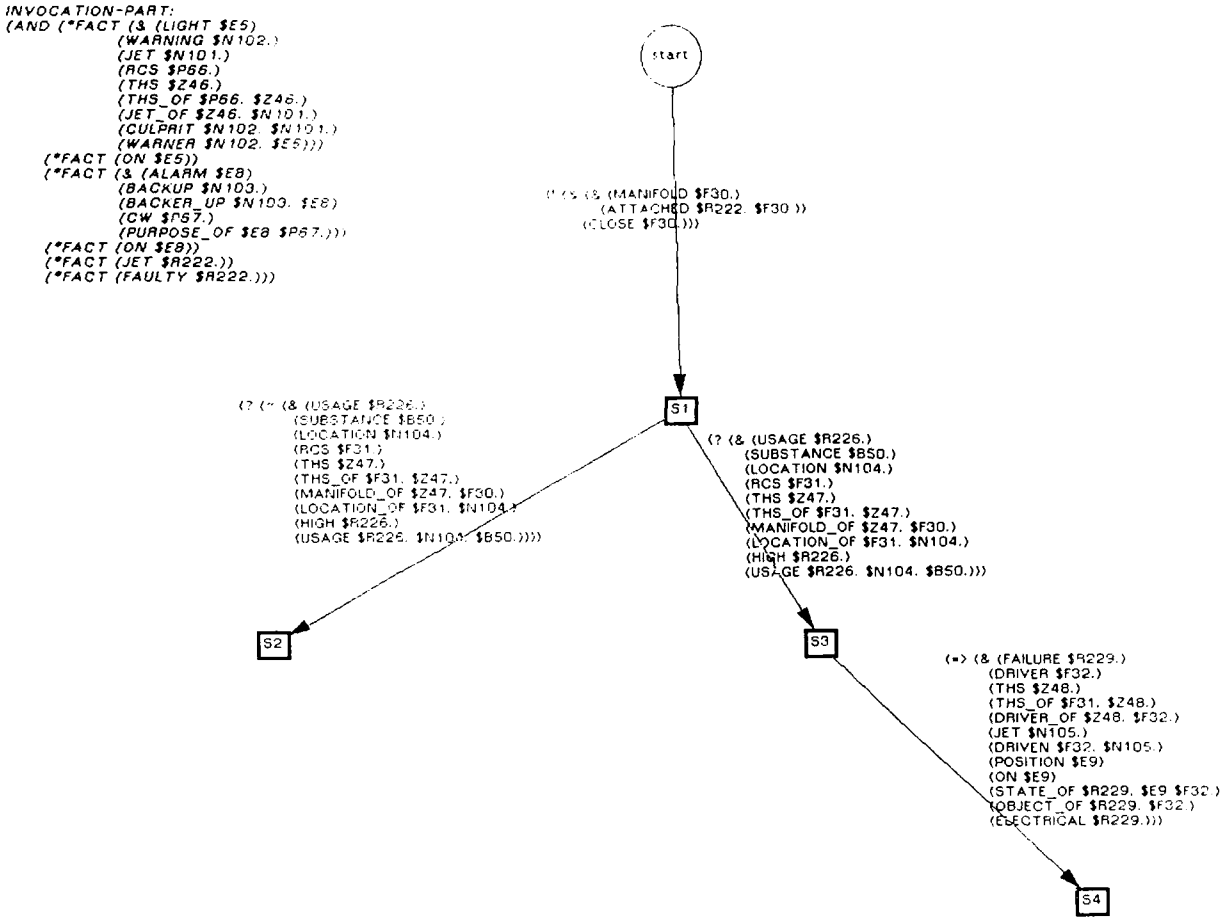


Figure 1.2: Candide's Architecture

number of improvements were made to it. Chapter 2 contains a detailed description of PATR-II; a User's Manual and sample session are presented in Appendix B.

One significant improvement made to PATR-II is the development of a new structure-sharing representation of directed graphs. In PATR-II, most of the information used during sentence analysis is represented by directed graphs of features and their values. The structure-sharing representation minimizes the amount of copying that the parser has to do in the course of interpreting sentences. For each word in the input sentence, the parser must create a copy of the word's lexical entry so that unifications with the entry do not affect the dictionary. Every time unification is invoked in the course of analysis, the affected graphs must be copied because the original graphs may still be needed for other parts of the analysis. Much of this copying can be avoided by making use of the structure-sharing technique, in which virtual copies of the graph share as much as possible of the original structure. The addition of structure-sharing to the system improved efficiency by approximately 20 percent. Chapters 3 and 4 of this document describe the structure-sharing mechanism.

Other advances made to PATR-II during the Candide project include the introduction of restriction mechanisms for parsing and the enhancement of the morphological analyzer. The definition of the restriction method for uniformly controlling the amount of top-down information used in parsing led to dramatic improvements in the efficency of the PATR-II implementation. A description of the method in a more general setting, including its incorporation into parsing algorithms for unification-based formalisms such as PATR-II, is taken up in Chapter 5. The morphological analyzer was completely overhauled during the term of this project. As a result it can directly use morphological rules instead of precompiled automata. This work on morphological analysis is discussed in Chapter 6.

Finally, work was done on the issue of grammar compilation. The current PATR grammar system is an interpreter. Grammar rules are represented quite directly as data structures, and these structures are interpreted in the course of parsing. This approach makes for a flexible grammar-writing environment, but is less efficient than a system that compiles grammar rules into directly executable code would be. A grammar compiler, P-PATR, was developed, which tranforms PATR grammars into Prolog programs, for which efficient compilers exist. P-PATR is a multipass grammar compiler that converts a PATR grammar, including its lexicon and lexical rules, into a Prolog program that parses input sentences according to a left-corner algorithm. Chapter 7 of this document discusses P-PATR in detail.

1.4 Semantic and Pragmatic Analysis

The second major component in the Candide system, CANDIDE-SPI, performs semantic and pragmatic interpretation. The primary input to this system consists of feature structures that encode the syntactic analysis of an utterance. In addition, CANDIDE-SPI produces representations of the discourse context during the processing of each utterance; and the discourse context for each utterance is always available during processing of the subsequent one. Intermediate representations are conditional interpretations, which consist of a partial interpretation (called the *sense*) plus a set of assumptions about subsequent pragmatic processing. Semantic interpretation rules operate in a top-down recursive fashion on the input feature structure, converting portions of it to partial interpretations under

assumptions of subsquent pragmatic processing. Separate pragmatic rules then discharge the assumptions, in the process further altering the interpretations. The pragmatic rules can also read from and write to the discourse context.

CANDIDE-SPI contains routines from handling a range of pragmatic phenomena, including pronoun resolution, definite reference, quantifier scoping, resolution of compound nominals, and resolution of syntactic ambiguity such as prepositional-phrase attachment. The system is also capable of handling interactions among these phenomena – a problem that has proved challenging for many interpretation systems. To support the various pragmatic processes, a set of routines for handling the discourse context and the system's knowledge base were developed. The discourse context comprises three components: an immediate context, representing both type and syntactic information about the entities referred to in the utterance currently undergoing processing; a local context, representing similar information about the entities referred to in the immediately preceding context segment; and a global context, representing just type information about entities referred to throughout the discourse. The latter is configured as a stack; in the testbed system for acquiring procedural nets, this configuration can be implicit since the discourse context allows us to make us of current theories of refernce such as the centering theory of pronominal resolution and the focus-based theory of definite reference resolution.

To solve pragmatic problems, CANDIDE-SPI makes use of two knowledge stores: a knowledge base and a semantic dictionary. The knowledge base comprises both a type hierarchy and a specification of the roles associated with any type. The encoding of role information included a specification of the algebraic relations between any entity and the entities that may fill some role of it: To resolve definite reference properly, it is important to know whether a role is a function or a relation of an entity of some type, and, if the former, whether or not it is invertible—if the latter, whether or not it is many-1. This information is encoded in the knowledge base. Relations among the role-fillers are also encoded, and used in the analysis of linguistic modification and ellipsis resolution.

The semantic dictionary contains information about selectional restrictions on verbs, relational nouns, verbal nouns, and prepositions, as well as prepositional coercion rules. As in the knowledge base, relations among the arguments of any vocabulary item can be encoded.

Details of the semantic and pragmatic interpretation module are given in Chapters 8 and 9 of this document.

References

[1] Georgeff, M P. and A.L. Lansky. A system for reasoning in dynamic domains: Fault diagnosis on the space shuttle Technical Note 375, Artificial Intelligence Center, SRI International, Menlo Park, California, 1985.

[2] Georgeff, M.P. and A.L. Lanksy. Procedural reasoning. *Proceedings of the IEEE, Special Issue on Knowledge Representation* (1986), 1383-1398.

[3] Lowrance, J.D. GRASPER II Reference Manual. SRI International Proprietary Document.

Chapter 2

The PATR-II Experimental System

This chapter was written by Stuart Shieber.

2.1 Introduction and History

The Natural Language Group of the Artificial Intelligence Center at SRI International has for many years been pursuing research on formalisms for encoding linguistic information for use by computers. One line of this research, beginning with the SRI Speech Understanding System [Paxton, 77], and continuing through the LIFER project [Hendrix, 77], LADDER [Hendrix, *et al.*, 78], D-LADDER [Konolige, 79], TED [Hendrix and Lewis, 81], and DIALOGIC [7], has its current incarnation in the PATR project. This project began three years ago under a charter to do research leading towards a declarative, mathematically well-founded reconstruction of DIALOGIC, but evolved into a project to design from scratch new grammar formalisms based on recent advances in linguistics and computer science. The first such formalism, called PATR, was designed and implemented by Stan Rosenschein and the author [23]. The current formalism, PATR-II, a radical departure from PATR, and even more radically from DIALOGIC, was designed by the author with various members of the PATR group, primarily Fernando Pereira, and Lauri Karttunen. Descriptions of the PATR-II formalism can be found in [12] and [15], the latter providing a brief description of the various previous implementations. Theoretical work on a denotational semantics for the formalism can be found in [10]. Related work on grammar formalisms based on the unification of directed graphs includes [Karttunen, 84] and [Wittenburg, 84].

Various parsing systems, grammar debugging environments, and question-answering systems based on PATR and PATR-II have been implemented. This report discusses the newest implementation of PATR-II, a ZETALISP implementation for the Symbolics 3600, which includes a grammar compiler, a grammar debugging environment (including incremental compilation of grammar rules, lexical entries, etc., tracing package, grammar editing package), and a parser based on a left-corner parsing algorithm.

This document provides a rough description of the PATR-II formalism and the operation and implementation of the PATR-II Experimental System. Appendix B provides a user manual for the system and protocol of a sample run of the system, manifested in an annotated series of snapshots of the screen.

2.2 The PATR-II Formalism

This section was derived from material prepared by the author for the Tenth International Conference on Computational Linguistics. It can be skipped by those uninterested in the linguistic motivation and usage of the formalism.

Building on a convergence of ideas from the linguistics and AI communities, PATR-II takes as its primitive operation an extended pattern-matching technique, *unification*, first used in logic and theorem-proving research and lately finding its way into research in linguistics [Kay, 79; Gazdar and Pullum, 82] and knowledge representation [Reynolds, 70; Ait-Kaci, 83]. Instead of unifying logic terms, however, PATR unification operates on directed acyclic graphs (DAG).[1]

DAGs can be atomic symbols or sets of label/value pairs, where the values are themselves DAGs (either atomic or complex). Two labels can have the same value—thus the use of the term *graph* rather than *tree*. DAGs are notated either by drawing the graph structure itself, with the labels marking the arcs, or, as in this paper, by notating the sets of label/value pairs in square brackets, with the labels separated from their values by a colon; e.g., a DAG associated with the verb "knight" (as in "Uther wants to knight Arthur") would appear (in at least one of our grammars) as

```
[cat: v
 head: [aux: false
        form: nonfinite
        voice: active
        trans: [pred: knight
                arg1: <f1134>
                      []
                arg2: <f1138>
                      []]]
 syncat: [first: [cat: np
                  head: [trans: <f1134>]]
          rest: [first: [cat: np
                         head: [trans: <f1138>]]
                 rest: <f1140>
                       lambda]
          tail: <f1140>]]
```

[1]Technically, these are rooted, directed, acyclic graphs with labeled arcs. Formal definition of these and other technical notions can be found in Appendix A of [15]. Note that some implementations have been extended to handle cyclic graph structures as well as graph structures with disjunction and negation [Karttunen, 84].

Reentrant structure is notated by labeling the DAG with an arbitrary label (in angle brackets), then using that label for future references to the DAG.

Associated with each entry in the lexicon is a set of DAGs. The root of each DAG will have an arc labeled *cat* whose value will be the category of the associated lexical entry. Other arcs may encode information about the syntactic features, translation, or syntactic subcategorization of the entry. But only the label *cat* has any special significance; it provides the link between context-free phrase structure rules and the DAGs, as explicated below.

PATR-II grammars consist of rules with a context-free phrase structure portion and a set of unifications on the DAGs associated with the constituents that participate in the application of the rule. The grammar rules describe how constituents can be built up to form new constituents with associated DAGs. The right side of the rule lists the *cat* values of the DAGs associated with the filial constituents; the left side, the *cat* of the parent. The associated unifications specify equivalences that must exist among the various DAGs and sub-DAGs of the parent and children. Thus, the formalism uses only one representation—DAGs—for lexical, syntactic, and semantic information, and one operation—unification—on this representation.

By way of example, we present a trivial grammar for a fragment of English with a lexicon associating words with DAGs.

$S \rightarrow NP\ VP$

$$<VP\ agr> = <NP\ agr>$$

$VP \rightarrow V\ NP$

$$<VP\ agr> = <V\ agr>$$

Uther:

$$\begin{aligned} <cat> &= np \\ <agr\ number> &= singular \\ <agr\ person> &= third \end{aligned}$$

Arthur:

$$\begin{aligned} <cat> &= np \\ <agr\ number> &= singular \\ <agr\ person> &= third \end{aligned}$$

knights:

$$\begin{aligned} <cat> &= v \\ <agr\ number> &= singular \\ <agr\ person> &= third \end{aligned}$$

This grammar (plus lexicon) admits the two sentences "Uther knights Arthur" and "Arthur knights Uther." The phrase structure associated with the first of these is:

[S [NP Uther] [VP [V knights] [NP Arthur]]]

The VP rule requires that the *agr* feature of the DAG associated with the VP be the same as (unified with) the *agr* of the V. Thus, the VP's *agr* feature will have as its value the *same* node as the V's *agr*, and hence the same values for the *person* and *number* features. Similarly, by virtue of the unification associated with the S rule, the NP will have the same *agr* value as the VP and, consequently, the V. We have thus encoded a form of subject-verb agreement.

Note that the process of unification is *order-independent*. For instance, we would get the same effect regardless of whether the unifications at the top of the parse tree were effected before or after those at the bottom. In either case, the DAG associated with, e.g., the VP node would be

```
[cat: vp
 agr: [person: third
       number: singular]]
```

These trivial examples of grammars and lexicons offer but a glimpse of the techniques used in writing PATR-II grammars, and do not begin to employ the power of unification as a general information-passing mechanism. Examples of the use of PATR-II for encoding much more complex linguistic phenomena can be found in Shieber *et al.* [83].

2.2.1 Templates and Lexical Rules

Clearly, the bare PATR-II formalism, as it was presented in this section, is sorely inadequate for any major attempt at building natural-language grammars because of its verbosity and redundancy. Efficiency of encoding was temporarily sacrificed in an attempt to keep the underlying formalism simple, general, and semantically well-founded. However, given a simple underlying formalism, we can build more efficient, specialized languages on top of it, much as MACLISP might be built on top of pure LISP. And just as MACLISP need not be implemented (and is not implemented) directly in pure LISP, specialized formalisms built conceptually on top of pure PATR-II need not be so implemented (although currently we do implement them directly through pure PATR-II). The effectiveness of this approach can be seen in the fact that at least a sizable portion of English syntax has been encoded in various experimental PATR-II grammars constructed to date. The syntactic constructs encoded include subcategorization of various complement types (*NP*s, $\overline{S}$s, etc.), active, passive, "there" insertion, extraposition, raising, and equi-NP constructions, and unbounded dependencies (such as Wh-movement and relative clauses). Other theory-dependent devices that have been modeled with PATR-II include head-feature percolation [Gazdar and Pullum, 82], and LFG-like semantic forms [Kaplan and Bresnan, 83]. Note that none of these constructs and techniques required expansion of the underlying formalism; indeed, the constructions all make use of the techniques described in this section. See Shieber *et al.* [83] for a detailed discussion of the modeling of some of these phenomena.

The devices now available for molding PATR-II to conform to a particular intended usage or linguistic theory are in their nascent stage. However, because of their great importance in making the PATR-II system a usable one, we will discuss them briefly. It is important

to keep in mind that these methods should not be considered a part of the underlying formalism, but merely "syntactic sugar" to increase the system's utility and allow it to conform to a user's intentions.

Templates

Because so much of the information in the PATR-II grammars under actual development tends to be encoded in the lexicon, most of our research has been devoted to methods for removing redundancy in the lexicon by allowing the users themselves to define primitive constructs and operations on lexical items. Primitive constructs, such as the transitive, dyadic, or equi-NP properties of a verb, can be defined by means of *templates*, that is, DAGs that encode some linguistically isolable portion of the DAG of a lexical item. These template DAGs can then be combined to build the lexical item out of the user-defined primitives.

As a simple example, we could define (with the following syntax) the template *Verb* as

Let *Verb* be

$$<cat> \;=\; V$$

and the template *ThirdSing* as

Let *ThirdSing* be

$$<agr\ number> \;=\; singular$$

unif <agr person> = third.

The lexical entry for "knights" would then be

knights:

Verb ThirdSing

Templates can themselves refer to other templates, enabling definition of abstract linguistic concepts hierarchically. For instance, a *modal verb* template may use an *auxiliary verb* template, which in term may be defined using the *verb* template above. In fact, templates are currently employed for abstracting notions of subcategorization, verb form, semantic type, and a host of other concepts.

Lexical Rules

More complex relationships among lexical items can be encoded by means of lexical rules. These rules, such as passive and "there" insertion, are user-definable operations on the lexical items, enabling one variant of a word to be built from the specification of another variant. A lexical rule is specified as a set of selective unifications relating an input DAG and an output DAG. Thus, unification is the primitive used in this device as well.

Lexical rules are used to encode the relationships among various lexical entries that would typically be thought of as transformations or relation-changing rules (depending on

one's ideological outlook). Because lexical rules perform these operations, the lexicon need include only a prototype entry for each verb. The variant forms can be derived through lexical rules applied in accordance with the morphology actually found on the verb. (The morphological analysis in the first ZETALISP implementation of PATR-II is performed by a program based on the system of Koskenniemi [83] and was written by Lauri Karttunen [83].)

For instance, given a PATR-II grammar in which the DAGs are used to emulate the f-structures of LFG, we might write a *passive* lexical rule as follows (following Bresnan [83]):[2]

Define *Passive* as

$$\begin{aligned} <out\ cat> &= <in\ cat> \\ <out\ form> &= passprt \\ <out\ subj> &= <in\ obj> \\ <out\ obj> &= <in\ subj> \end{aligned}$$

The rule states in effect that the output DAG (the one associated with the passive verb form) marks the lexical item as being a passive verb whose object is the input DAG's subject and whose subject is the input's object. Such lexical rules have been used for encoding the active/passive dichotomy, "there" insertion, extraposition, and other so-called relation-changing rules.

2.3 The PATR-II Experimental System

The PATR-II Experimental System is a natural-language processing system based on the PATR-II formalism just presented. The basic operations that are performed by the system are itemized below, with brief descriptions and references to the implementing code.[3] More detailed descriptions of some of these basic types of operations are presented in the subsections below.

- *Grammar compiling.* The system compiles grammars stored in text files in the format presented above. A deterministic top-down recursive-descent "meta-parser" (i.e., a parser for the metalanguage, PATR-II, as opposed to a parser for the object language, say, English) parses the files and compiles the grammar rules, templates, lexical rules, and lexical entries into tables used by the rest of the system. Code for this purpose is found in the file PATR-LOAD.LISP.

- *Grammar maintenance.* The system stores information about grammar rules, words, word senses, templates and lexical rules using an object-oriented style. (The Flavors system in ZETALISP is an object-oriented programming package used extensively in the implementation as it provides the best method of information-hiding and encoding

[2]The example is merely meant to be indicative of the syntax for and operation of lexical rules. We do not present this as a valid definition of *Passive* for any grammar we have written in PATR-II.

[3]The top-level functions organizing all of these operations are found in PATR-MAIN.LISP.

abstract data types such as dag, rule, etc.) Code for this purpose can be found in the files PATR-RULE.LISP, PATR-WORD.LISP, PATR-NONTERMINAL.LISP.

The system maintains information about the location of source information for PATR rules[4]. An interface (PATR-EDIT.LISP) with the ZMACS editor allows editing of individual rules and incremental compilation of PATR rules. The ability to display particular rules, words, word senses, etc. is also provided for the user. (See Appendix B for further details.)

- *Sentence parsing.* Sentences typed at top level are parsed by a bottom-up left-corner chart parser. Code for this purpose is found in the files PATR-LC.LISP and PATR-VERTEX.LISP. The chart is encoded using flavors; see PATR-EDGE.LISP and PATR-VERTEX.LISP. A tracing package (PATR-TRACE.LISP) allows the tracing of some or all of the grammar rules, so that information is printed as parsing occurs concerning the active and passive edges involving those rules. (See Appendix B for further details.)

- *Chart perusal.* After parsing, the chart is available for perusal by the user. Listings of edges coming into particular chart vertices, detailed listing of edge information (including failed edges), etc. are all available through a mouse-oriented interface (PATR-MAIN.LISP).

- *System configuration.* The profile package (PATR-PROFILE.LISP) allows some dynamic reconfiguration of the system. By setting options in the profile menu, the parser can be reconfigured to use or not use top-down filtering, to allow epsilon rules in grammars, to incorporate special unifications for filler-gap dependency information, the tracing package can be turned on or off, as can the storing of failed edges in the chart, loading (compiling) of files can be configured to display each rule loaded or not, and to display each token read or not, etc. (See Section B.2.1 for further details.)

In addition, the following utilities are provided.

- *System maintenance.* The entire PATR-II system is maintained using a system definition found in PATR-SYSTEM.LISP.

- *DAG Manipulation.* A DAG package (PATR-DAG.LISP) provides a complete utility for creating, unifying, manipulating, and displaying directed acyclic graph structures.

- *Printing, formatting and input/output utilities.* Extra utilities for handling fonts and printing mouse-sensitive items can be found in PATR-FORMAT.LISP.

We now present more detailed descriptions of some of these operations, concentrating on the data representation and algorithms used.

[4]The term PATR rule is used to include grammar rules, templates, lexical rules and lexical entries.

2.3.1 Grammar Compiling

Format of PATR-II grammar files

PATR-II grammar files can include grammar rules, template and lexical rule definitions, lexical entries, and input file specifications. All but the last of these have been discussed in Section 2.2. The format for files with these rules is almost identical to that presented above. The commenting conventions of ZETALISP are obeyed (i.e., anything after a semicolon and until the next carriage return is a comment, and anything delimited by "#|" and "|#" is a comment). The mode line convention is also obeyed, so that files not ending in the extensions ".lex", ".defs", and ".gram" can still be put into PATR mode in the editor.

We discuss each type of rule separately.

- *Grammar rules.* The format for a grammar rule is the following. The rule is introduced by the keyword "Rule" followed by a unique identifying phrase enclosed in braces. Then the context-free phrase structure portion of the rule is given, followed by a colon. (The arrow symbol is obtained with symbol-k on the 3600 keyboard.) Finally, a list of unifications is given using the angle bracket notation. Grammar rules (and all rules) are ended by periods. There are no formatting restrictions on where spaces can occur. They are only required to separate tokens in the obvious places.

  A sample rule might be

  ```
  Rule {generic sentence formation}
      S → NP VP
          <S head> = <VP head>
          <NP head agr> = <VP head agr>.
  ```

- *Templates.* The format for template definitions is exactly that presented in Section 2.2.1. E.g.,

  ```
  Let ThirdSing be
      <agr number> = singular
      <agr person> = third.
  ```

- *Lexical rules.* Again, the format is exactly as presented above (Section 2.2.1). E.g.,

  ```
  Define Passive as
      <out cat> = <in cat>
      <out form> = passprt
      <out subj> = <in obj>
      <out obj> = <in subj>.
  ```

- *Lexical entries.* Here, some differences with the notation in Section obtain. Lexical entries are introduced by the keyword "Word" followed by the word itself, some unifications (and template or lexical rule names), and then variants of the word, i.e., word senses are given. Each word sense is given as further unifications to perform on the word's dag. Each variant is set off from the preceding part by a hyphen. Thus an entry might be:

  Word knighted
  Verb ThirdSing
  - PastTense Active
  - Passive.

  for a word "knighted" with two variants, one given by "Verb ThirdSing PastTense Active" and one by "Verb ThirdSing Passive". The full PATR-II implementation incorporates a morphological analyzer written by John Bear.

- *Input file rules.* Finally, the file can contain instructions to load other files. The format for these is to precede the file name (or path name) enclosed in double quotes by the keyword "Input". The rule, as usual, is delimited by a period. E.g.,

  Input ">patr>english.gram".

  This would cause the contents of the file >PATR>ENGLISH.GRAM to be loaded at this point before continuing.

Parsing of grammar files

The parser of PATR-II files is a deterministic, top-down, recursive-descent parser written directly in ZETALISP using the standard techniques for recursive-descent parsing. Deterministic behavior is achieved through one token lookahead, which is sufficient for the LL(1) grammar for PATR-II. The interface to lexical analysis is through the function GET-TOKEN which updates variables CURRENT-TOKEN and NEXT-TOKEN to the currently scanned and lookahead tokens respectively. NEXT-TOKEN is used to guide the parse, CURRENT-TOKEN to check that the syntax is being appropriately followed. Error recovery is achieved through a THROW to a resynchronization routine (SKIP-TO-NEXT-RULE) that merely looks for the end of the current rule being scanned to resynchronize.

A full LL(1) grammar for the PATR-II format is given in PATR-LOAD.LISP.

2.3.2 Grammar Maintenance

While loading the grammar, the information in the grammar is compiled into a set of data structures for use by the parser and for the user to interact with. The encoding of grammar information is done using the ZETALISP Flavor system because it allows the nearest thing

to data abstraction available in ZETALISP. Thus, there are flavors for rules, templates, lexical rules, and words, which hold the information about each of these types of objects. Furthermore, a set of tables is built up during grammar-loading which are later used by the parsing algorithm. These include tables for three relations: the left-corner relation, the first relation, and the first^{-1} relation.

The left-corner relation is merely a relation between nonterminals and the rules which have that nonterminal as left corner. When loading a rule, a hash table indexed by nonterminal is updated to add the rule storing it under its left corner.

The first relation is a relation between a nonterminal and all of the nonterminals that can occur underneath the given nonterminal along a left branch. The first^{-1} relation is roughly the converse, relating nonterminals and those nonterminals appearing above it and along a left branch. The first and first^{-1} relations are computed simultaneously, though only the former is used in parsing. This is because computation of the two is mutual. The technique, based on an earlier implementation by John Bear, works as follows:

Given a rule with left-hand nonterminal A and left-corner L, we update the first relation by adding to the first relation for A and all members X of first^{-1}(A) the nonterminals L, first(X), and first(L). Conversely, we update the first^{-1} relation by adding to it for L and all members X of first(L) the nonterminals A, first^{-1}(X) and first^{-1}(A).

The relations are all encoded as hash tables indexed by nonterminal, with values being the list of nonterminals in the relation under question with the given nonterminal.

2.3.3 Sentence Parsing

The PATR-II Experimental System uses a bottom-up, left-corner parsing algorithm to parse the object language of the PATR-II grammars. The algorithm uses top-down filtering and early evaluation of unifications to limit the amount of work done. This algorithm is the heart of the system and will be described in some detail. Implementing code is found in PATR-LC.LISP and PATR-VERTEX.LISP.

Motivation for the parsing algorithm

Motivation for the particular parsing algorithm chosen is based on experience with parsing English language constructs. The left-corner aspect of the algorithm was motivated by the SVO structure of English, with its concomitant prepositional tendencies, which lends itself to identifying handles by their left corner. (More radical types of parsing algorithms, loosely based on LR techniques are being worked on presently. See [Shieber, 83] for motivation for this approach to parsing natural languages.

Top-down filtering is an attempt to increase the utilization of top-down information in guiding the basically bottom-up operation of the parser. It keeps track of what categories of constituent are allowed to start at each vertex, and filters all edges that are attempted to be added which are not so allowed. Preliminary, and very limited, testing showed that as many as a third of the edges can be eliminated in this way.

Evaluation of unifications is done as early as possible to achieve the maximal effect of filtering out ill-formed edges, both passive and active. (Cf. other implementations of unification-based grammar formalisms, e.g., the Xerox LFG system, which postpone all unifications until after context-free parsing.)

The parsing algorithm

The parsing algorithm uses a *chart* to retain information about partial and complete constituents built during the parse. The chart is organized into vertices and edges connecting the vertices. Information about constituents is encoded in *edges* in the chart--*active edges* corresponding to partial constituents and *passive edges* corresponding to complete constituents. Edges are similar to the dotted rules of Earley's algorithm; they have a start and end vertex, a left-hand-side nonterminal and a sequence of right-hand-side nonterminals, some (or all or none) of which have already been found. For active edges, not all have been found, and the next one in the sequence that has yet to be found is referred to as the *need* of the edge. The algorithm works by adding a passive edge to the chart for each lexical item in the sentence (actually for each sense of each lexical item). Initially, the top-down filter allows only the start symbol and all nonterminals in first(start symbol) at the initial vertex.

The procedure for adding a passive edge p works as follows:

1. *Top-down filtering:* If the lhs of p is disallowed (by the top-down filter) to start at its start vertex then stop.
2. *Addition step:* Otherwise, add p to the chart as a passive edge from its start vertex to its final vertex.
3. *Prediction step:* For every rule whose left corner nonterminal is the nonterminal of p, set up a new active edge using this rule with no constituents found, and mark it as starting at the final vertex of p. Associate with this edge a *copy* of the DAG implicit in the unifications in the rule. We will unify the various constituents into this DAG as they are found. These are the so-called *hypothesis edges.*
4. *Extension step 1:* Find all the active edges whose final vertex is the start vertex of p and whose need is the nonterminal of p. These are the so-called *extendable edges.*
5. *Extension step 2:* For each edge in the hypothesis and extendable active edges, extend the edge by adding in p as a subconstituent of the edge. This process forms several new (passive and active) edges which are recursively added to the chart. Extending an active edge in this way involves immediately unifying in a copy of the DAG associated with the new constituent. Thus unifications are evaluated as soon as possible, even before the passive edge is built, and certainly before the sentence parse is completed, as is done in other systems.

Addition of active edges is performed by the following algorithm:

1. *Top-down filtering:* If the top-down filter disallows edges with this lhs, stop. It is never added to the chart.

2. *Addition step:* Otherwise, add the active edge to the chart.

3. *Update top-down filter:* Let n be the need of a. For each nonterminal in $n \cup \text{first}(n)$, allow that nonterminal to start at the final vertex of a. (I.e., top-down filtering will now allow edges with lhs n or any element of first(n) to start at this edge).

Since the basic operations used by the algorithm involve retrieving active edges with a given final vertex and a given need, active edges are stored with their final vertices (using the flavor VERTEX) and are indexed by their need using association lists. Note that the algorithm makes use of the left-corner, and first relation information precomputed by the grammar-loading functions (Section 2.3.2).

2.3.4 DAG Manipulation

The DAG manipulation package handles the building, printing, copying, and, most importantly, unifying of DAGs. The unification of DAGs is the prime operation in PATR-II. The algorithm we use, though not identical to the standard algorithm for term unification, is similar in its operation and its simplicity. Before discussing the unification algorithm, however, we describe the implementation of the DAG flavor.

Each node in a DAG is an instance of the DAG flavor, and, as such has several instance variables (corresponding to fields in a record, say) which encode various pieces of information. If the DAG is atomic, the node has an ATOM-NAME field. If the DAG is not atomic, it has an encoding of the arcs emanating from the node, the SUB-DAGS field, encoded as an alist associating to each label the node at the end of the arc, itself an instance of the DAG flavor. If the DAG has been in a unification which has failed, the FAILED? field is set and the FAILURE-POINTER is given the node with which unification was attempted. This information is used for debugging grammars to help locate what parts of the grammar caused unification failure. Finally, a field INVISIBLE-POINTER is used to force redirection from one node to another, so that all references to the node get interpreted as references to the node pointed to. It is through these invisible pointers that unification can work.

The unification algorithm

The unification algorithm works as follows. To unify DAG1 and DAG2:

1. Follow all invisible pointers on the two DAGs.

2. If the DAGs are the same, (i.e., have already been unified), then succeed.

3. Otherwise, if both DAGs are atomic, check that their atom-names are identical. If so, succeed; if not, fail.

4. Otherwise, if the first DAG (DAG1) is atomic, then unify DAG2 and DAG1. (This guarantees that the second DAG will be atomic.)

5. Otherwise, if the second DAG is atomic (and recalling that the first DAG is not), if the first DAG has no arcs emanating from it, then make it into an atomic DAG with the same name as DAG2. Otherwise, fail, since atomic and complex DAGs cannot be unified.

6. Otherwise, neither DAG is atomic. We must unify each corresponding sub-DAG of the two DAGs, put the union of the arcs thus formed on the second DAG and redirect the first DAG to the second (via an invisible pointer). From then on all references to either DAG end up accessing the second DAG (as we would expect after unification). Of course, if any subunification fails, the whole unification fails as well.

The critical aspect of the algorithm is that all unified nodes are redirected to a common node. Thus all later references to one are references to the other as well.

The copying algorithm

Note that in the parsing algorithm presented, much use is made of the copying of DAGs so that unifications on two paths of the parsing do not interact. The copying algorithm is straightforward, though some care must be taken to maintain in the copy the unification links in the original DAG.

The DAG copying algorithm is as follows:

1. Follow all invisible pointers.

2. If the DAG has a value for the COPY-POINTER field, return that value.

3. If the DAG is atomic, make a new atomic DAG, store it in the *copy-pointer* field and return it.

4. Otherwise, the DAG is complex. Get a new DAG, and place it in the COPY-POINTER field of the original. For each arc in the original DAG, copy the sub-DAG at the end of the arc and store the copy of the sub-DAG under the same label in the copy of the original.

After a copy has been performed, the original DAG must be completely traversed to remove all the values in the COPY-POINTER field in case it is ever again copied.

2.4 References

References

[Ait-Kaci, 83] Ait-Kaci, H., 1983: "A new Model of Computation Based on a Calculus of Type Subsumption," Doctoral Dissertation Proposal, Dept. of Computer and Information Science, University of Pennsylvania (November).

[Bresnan, 83] Bresnan, Joan, 1983: *The mental representation of g. mmatical relations* (ed.), Cambridge: MIT Press.

[Gazdar and Pullum, 82] Gazdar, G. and G.K. Pullum, 1982: "GPSG: A Theoretical Synopsis," Indiana University Linguistics Club, Bloomington, Indiana.

[Grosz, *et al.*, 82] Grosz, B., N. Haas, G. Hendrix, J. Hobbs, P. Martin, R. Moore, J. Robinson and S. Rosenschein, 1982: "DIALOGIC: a core natural-language processing system," *Proceedings of the Ninth International Conference on Computational Linguistics*, Prague, Czechoslavakia (July), pp. 95-100.

[Hendrix, 77] Hendrix, G.G., 1977: "The LIFER Manual: A Guide to Building Practical Natural Language Interfaces," Technical Note 138, Artificial Intelligence Center, SRI International, Menlo Park, California (February).

[Hendrix and Lewis, 81] Hendrix, G.G., and W.H. Lewis, 1981: "Transportable Natural-Language Interfaces to Databases," Technical Note 228, Artificial Intelligence Center, SRI International, Menlo Park, California (30 April).

[Hendrix, *et al.*, 78] Hendrix, G.G., E.D. Sacerdoti, D. Sagalowicz, and J. Slocum, 1978: "Developing a Natural Language Interface to Complex Data," *ACM Transactions on Database Systems*, Volume 3, Number 2 (June).

[Kaplan and Bresnan, 83] Kaplan, R. and J. Bresnan, 1983: "Lexical-Functional Grammar: A Formal System for Grammatical Representation," in J. Bresnan (ed.), *The mental representation of grammatical relations* (ed.), Cambridge: MIT Press.

[Karttunen, 84] Karttunen, L., 1984: "Features and Values," in *Proceedings of the Tenth International Conference on Computational Linguistics*, Stanford University, Stanford, California (2-7 July, 1984).

[Karttunen, 83] Karttunen, L., 1983: "KIMMO: a general morphological processor," *Texas Linguistic Forum*, Volume 22 (December), pp. 161-185.

[Kay, 79] Kay, M., 1979: "Functional Grammar," in *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistics Society*, Berkeley, California (17-19 February).

[Kay, 83] Kay, M., 1983: "Unification Grammar," unpublished memo, Xerox Palo Alto Research Center.

[Konolige, 79] Konolige, K., 1979: "A Framework for a Portable Natural-Language Interface to Large Data Bases," Technical Note 197, Artificial Intelligence Center, SRI International, Menlo Park, California (12 October).

[Koskenniemi, 83] Koskenniemi, K., 1983: "A Two level Model for Morphological Analysis and Synthesis," forthcoming Ph.D. dissertation, University of Helsinki, Helsinki, Finland.

[Paxton, 77] Paxton, W.H., 1977: "A Framework for Speech Understanding," Technical Note 142, Artificial Intelligence Center, SRI International, Menlo Park, California (June).

[Pereira and Shieber, 84] Pereira, F. and S. Shieber, 1984: "The Semantics of Grammar Formalisms Seen as Computer Languages," in *Proceedings of the Tenth International Conference on Computational Linguistics*, Stanford University, Stanford, California (2-7 July).

[Reynolds, 70] Reynolds, J., 1970: "Transformational Systems and the Algebraic Structure of Atomic Formulas," in D. Michie (ed.), *Machine Intelligence*, Vol. 5, Chapter 7, Edinburgh, Scotland: Edinburgh University Press, pp. 135–151.

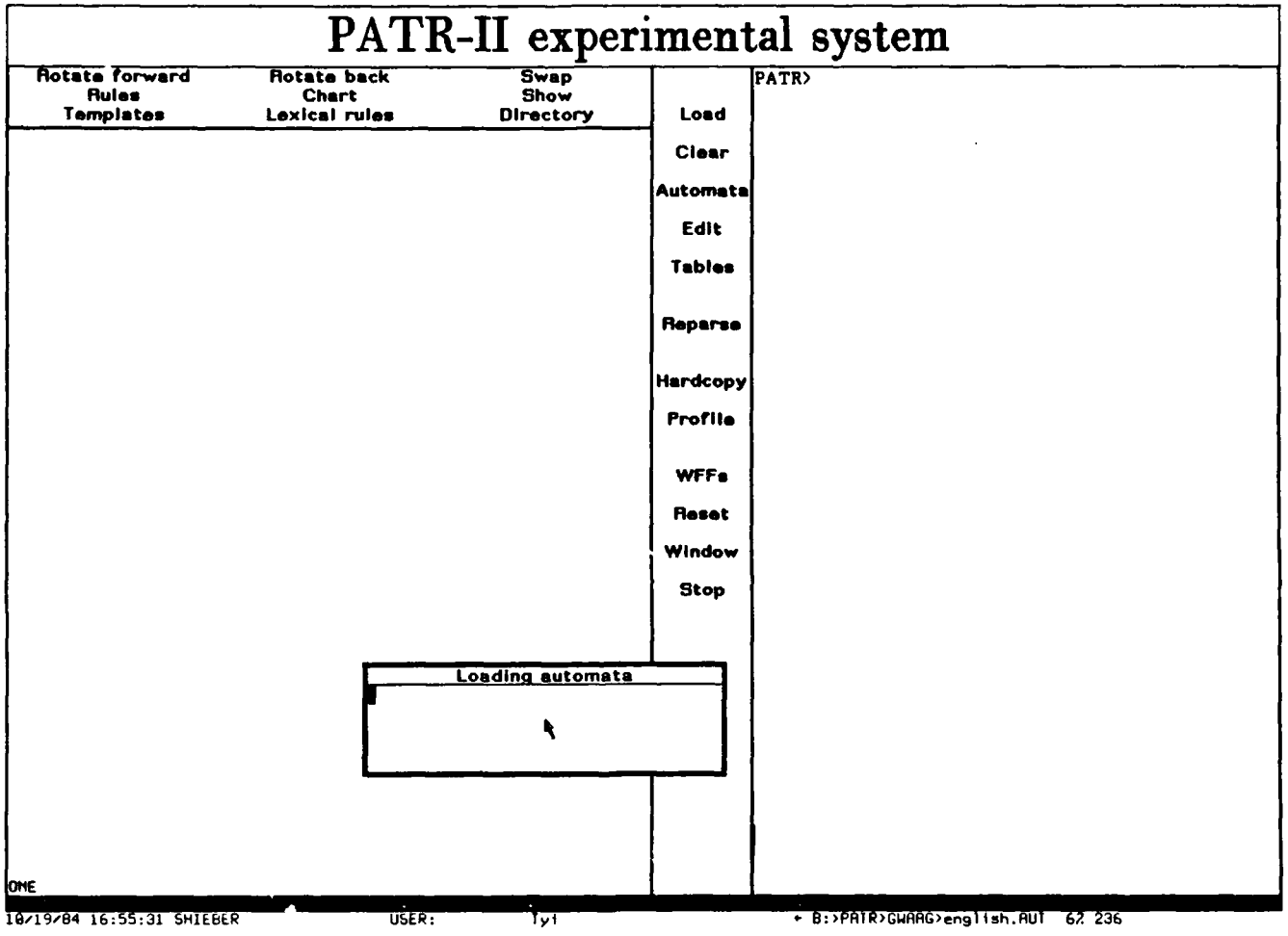
[Rosenschein and Shieber, 82] Rosenschein, S.J., and S.M. Shieber, 1982: "Translating English Into Logical Form," in *Proceedings of the Twentieth Annual Meeting of the Association for Computational Linguistics*, University of Toronto, Toronto, Ontario, Canada (16-18 June).

[Shieber, *et al.*, 83] Shieber, S., H. Uszkoreit, F. Pereira, J. Robinson, and M. Tyson, 1983: "The Formalism and Implementation of PATR-II," in B. Grosz and M. Stickel, *Research on Interactive Acquisition and Use of Knowledge*, SRI Final Report 1894, SRI International, Menlo Park, California (November).

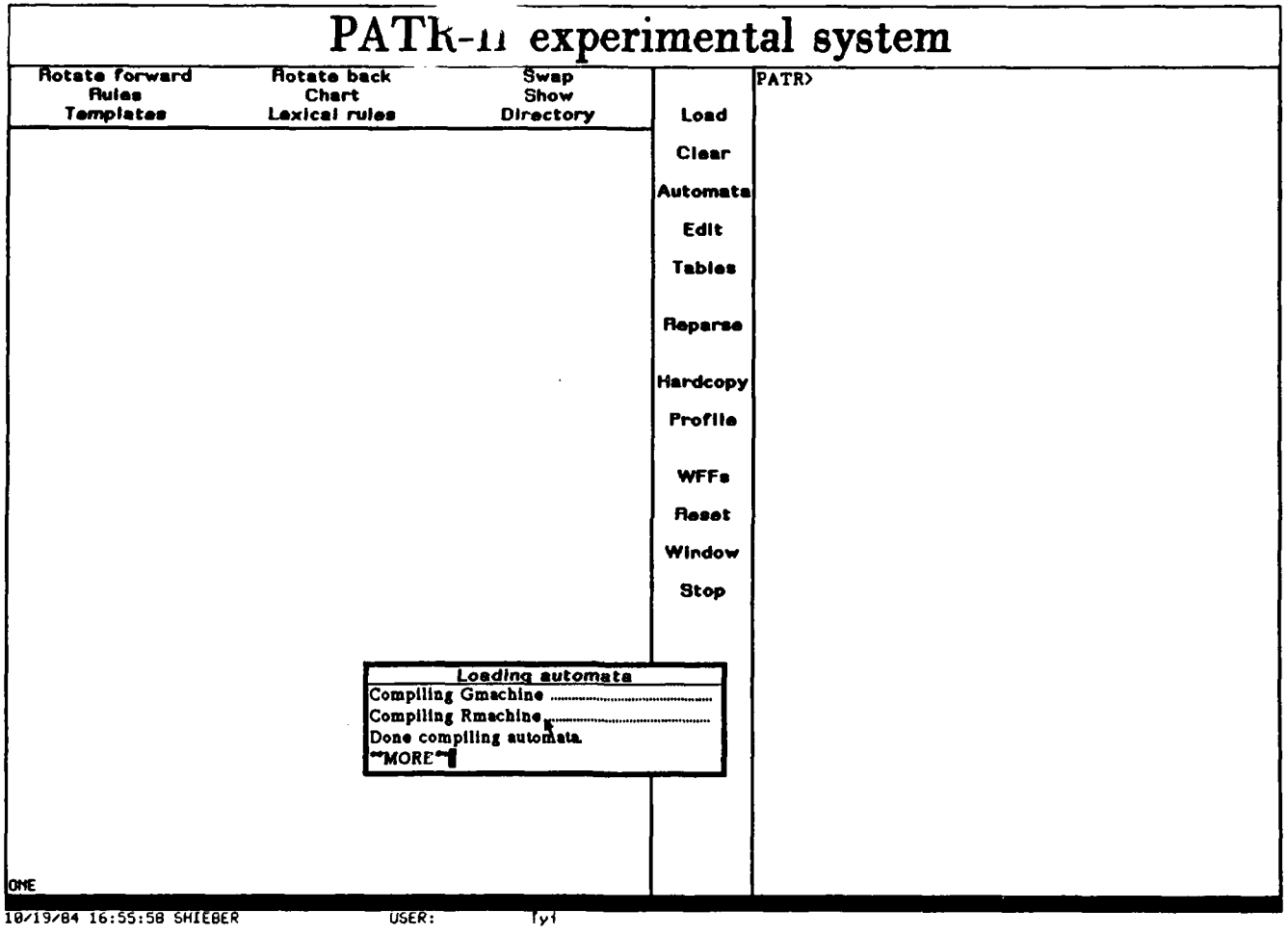
[Shieber, 83] Shieber, S. M., 1984: "Sentence Disambiguation by a Shift-Reduce Parsing Technique," in *Proceedings of the eignth International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany (8-12 August).

[Shieber, 84] Shieber, S.M., 1984: "The Design of a Computer Language for Linguistic Information," in *Proceedings of the Tenth International Conference on Computational Linguistics*, Stanford University, Stanford, California (2-7 July).

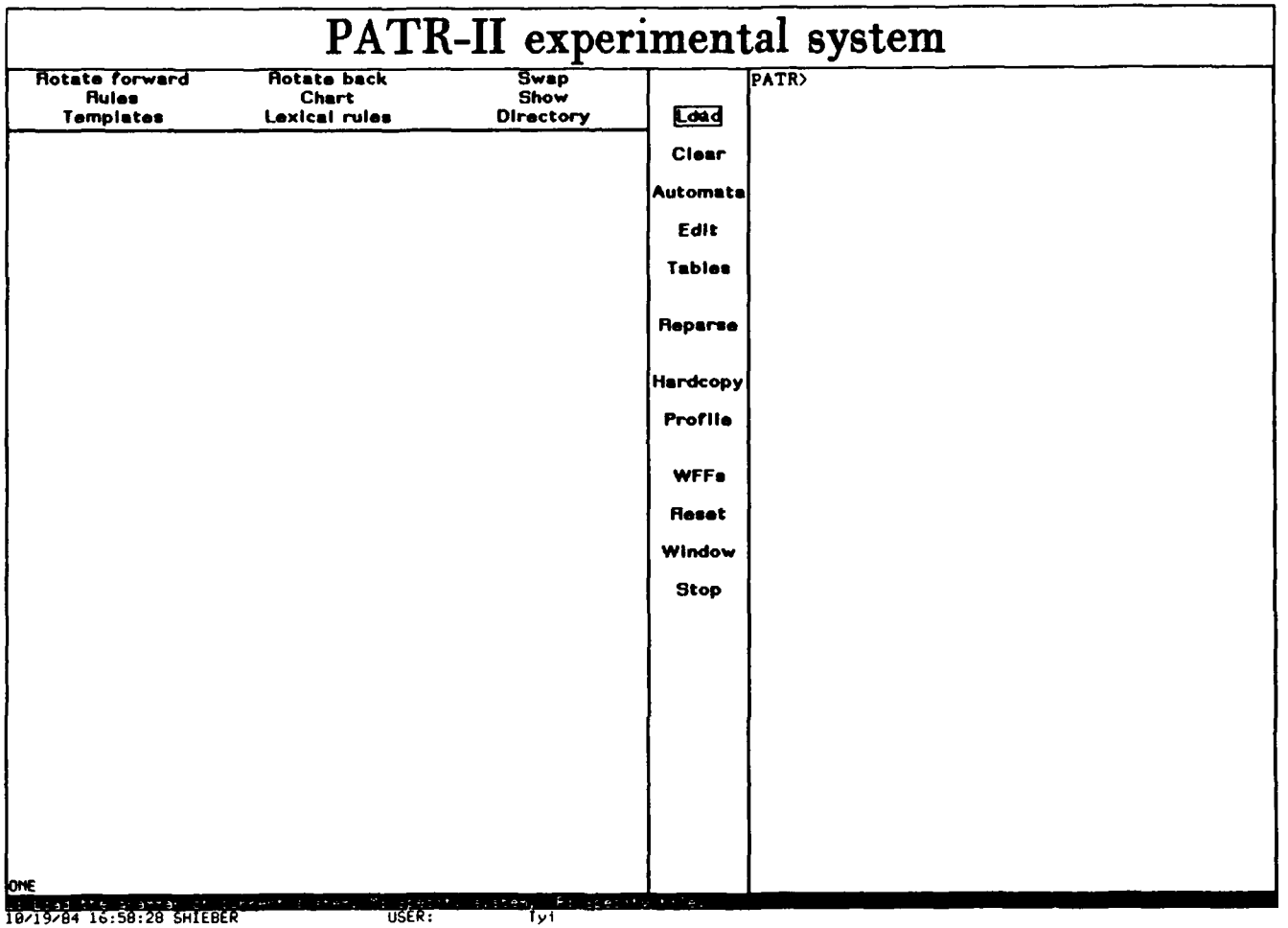
[Wittenburg, 84] "Unification Parsing with Lexical Rules as Disjunctive Acyclic Graphs," unpublished manuscript, Department of Linguistics, University of Texas, Austin, Texas (12 June).



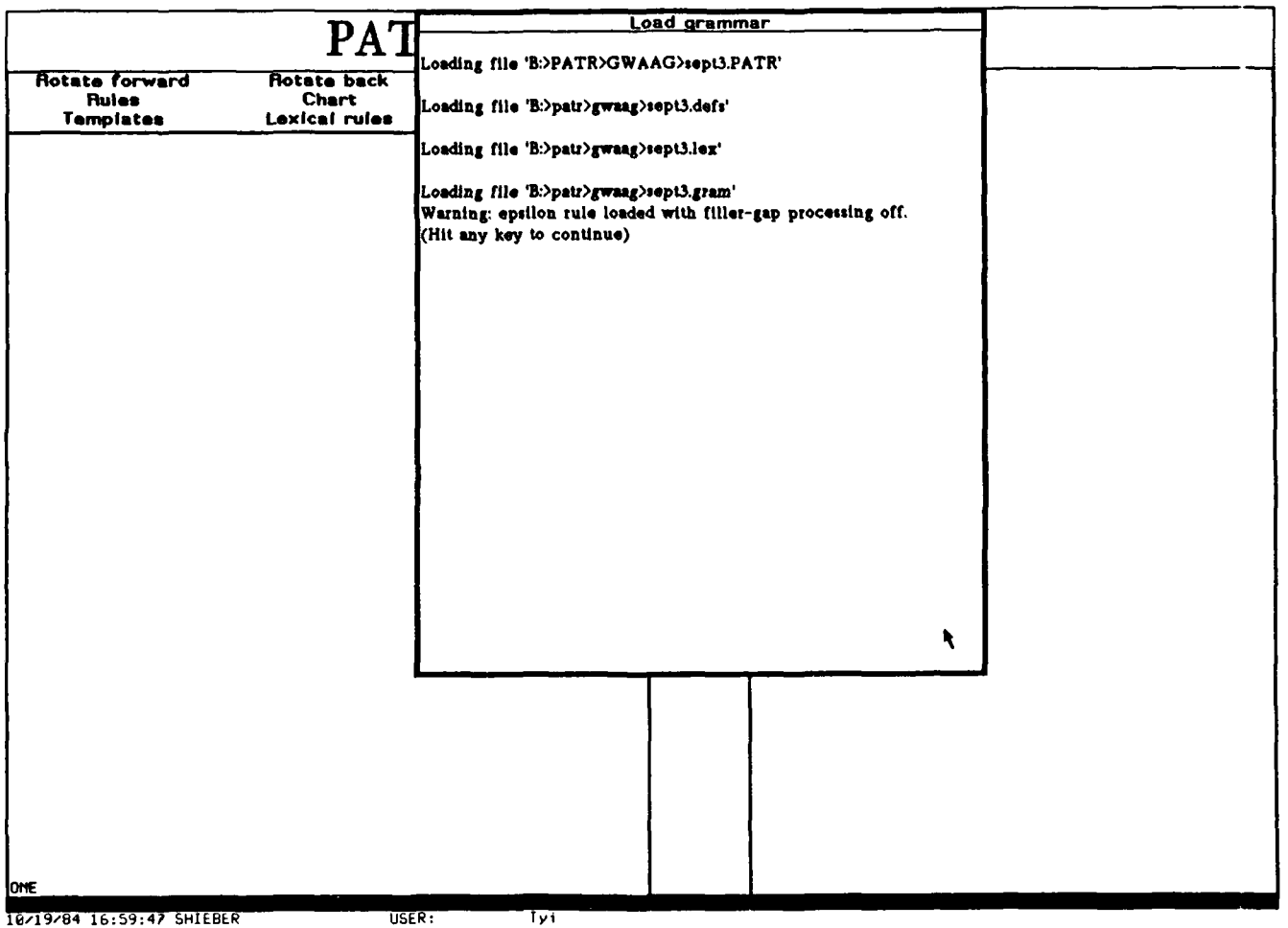
1. The system has just been started and the morphological analyzer automata are being loaded. Notification of this process is being done in a temporary window.



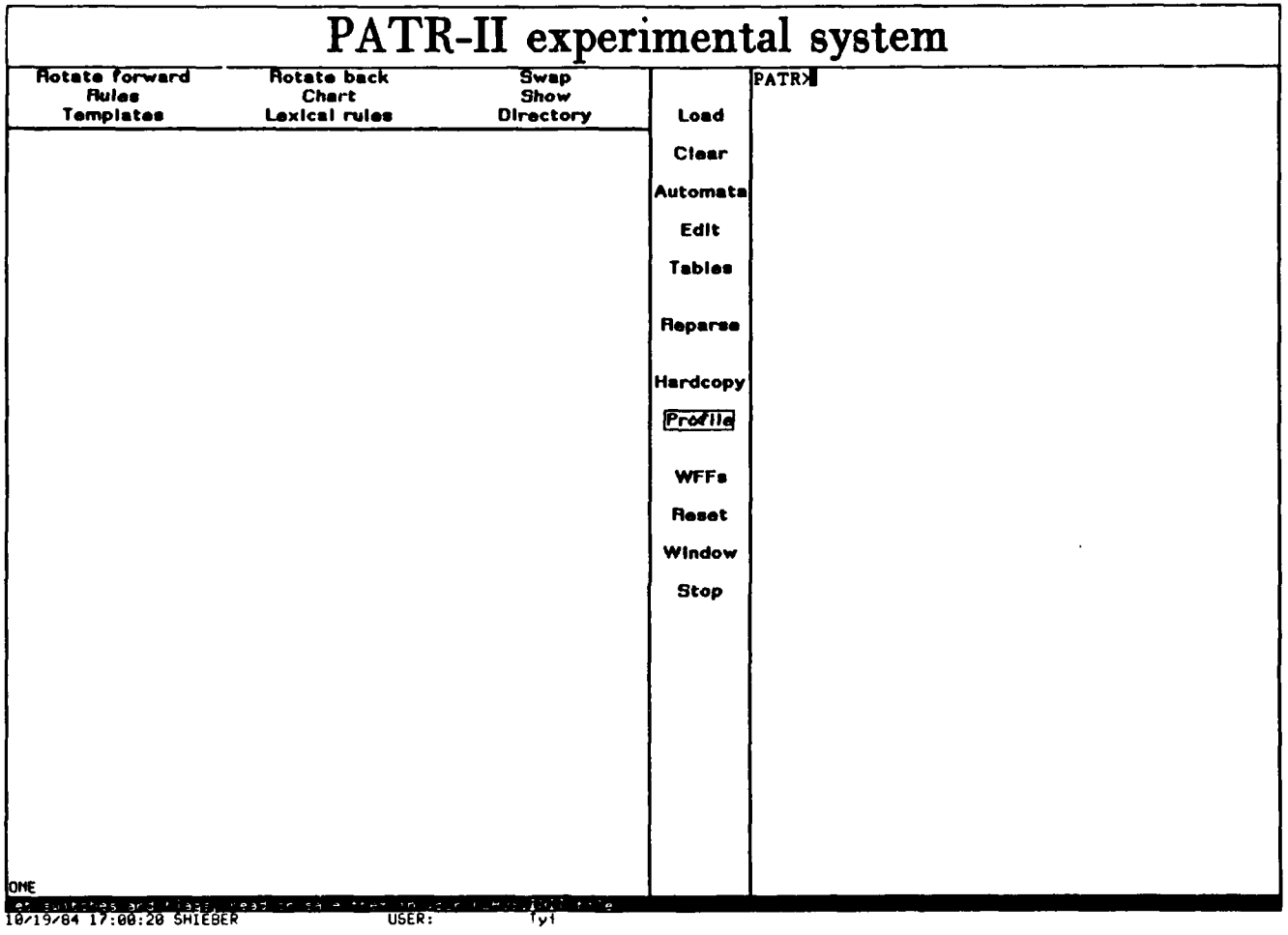
2. The automata loading is finished.



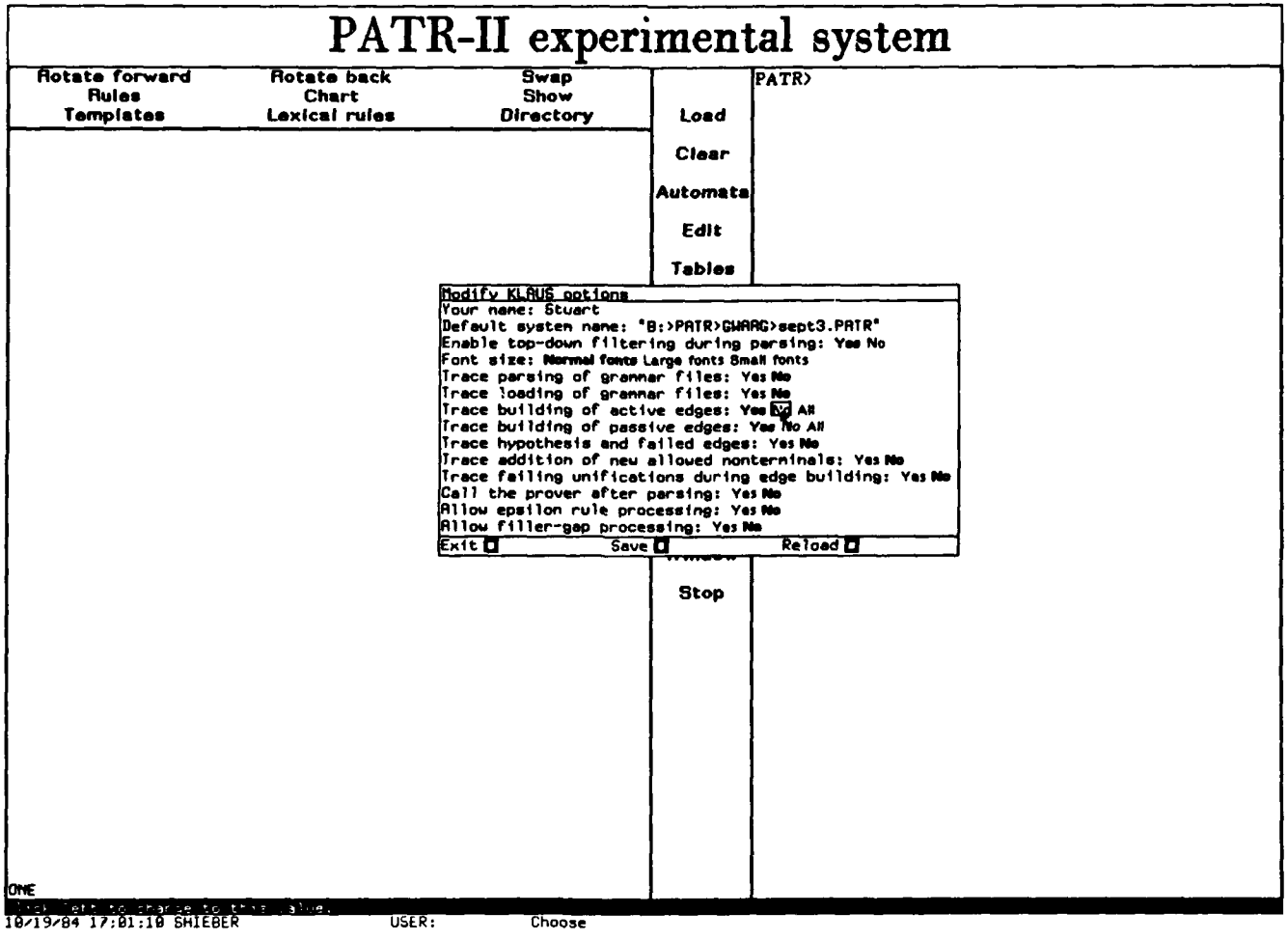
3. The system is now in a nascent state. The user is about to click left on "Load". Note location of mouse cursor, marked with an "x". The command being picked out is highlighted with a surrounding box. Note also that documentation concerning the command is displayed in the inverse video "mouse line" at the bottom of the screen. This is in general the case when the mouse cursor is over a command in a menu or a mouse-sensitive icon.



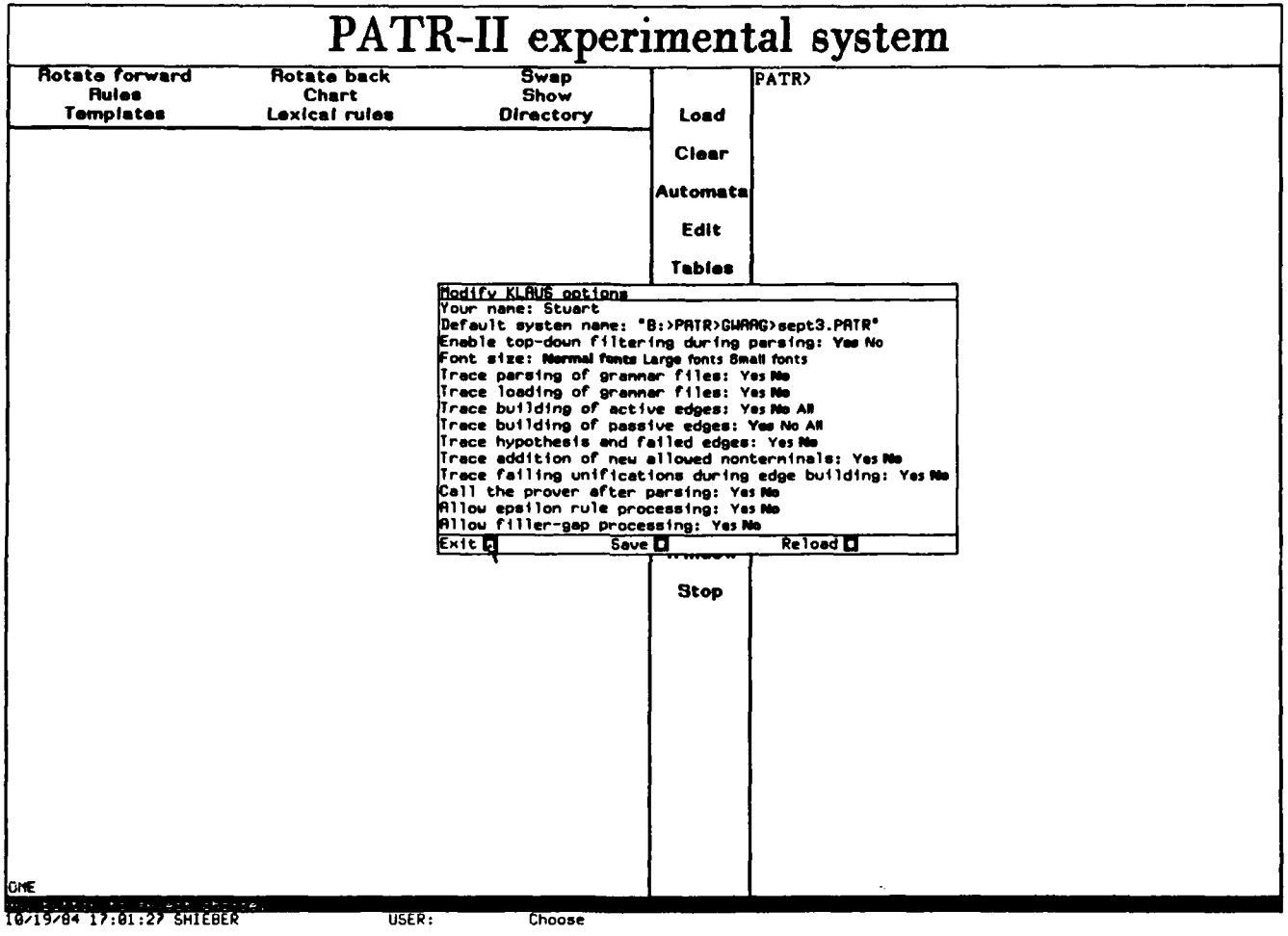
4. Grammar loading has completed. Notifications occurred in a temporary window. The system is now ready to parse sentences.



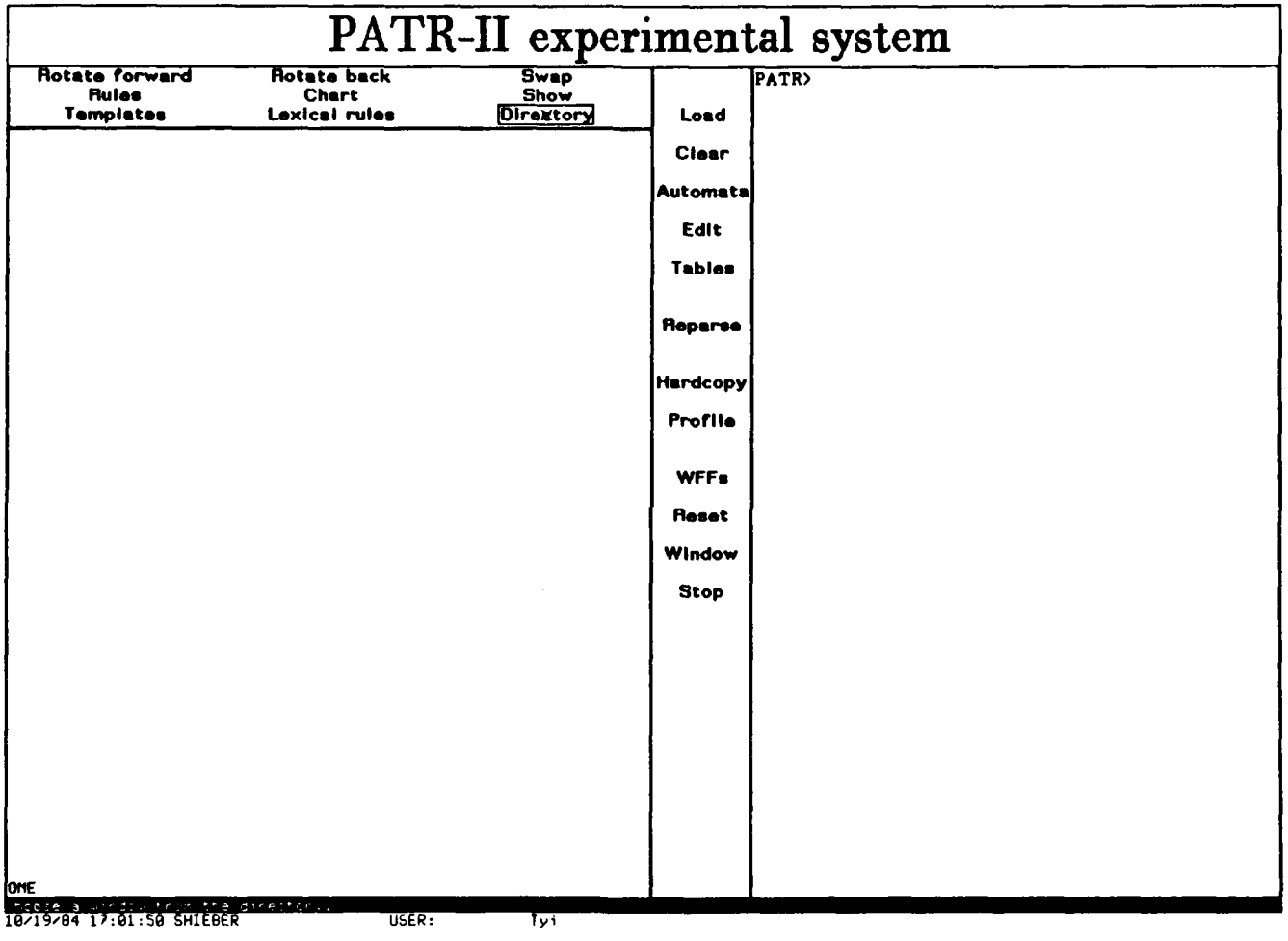
5. The user wants to change one of the properties of the system configuration, and so is about to click on "Profile".



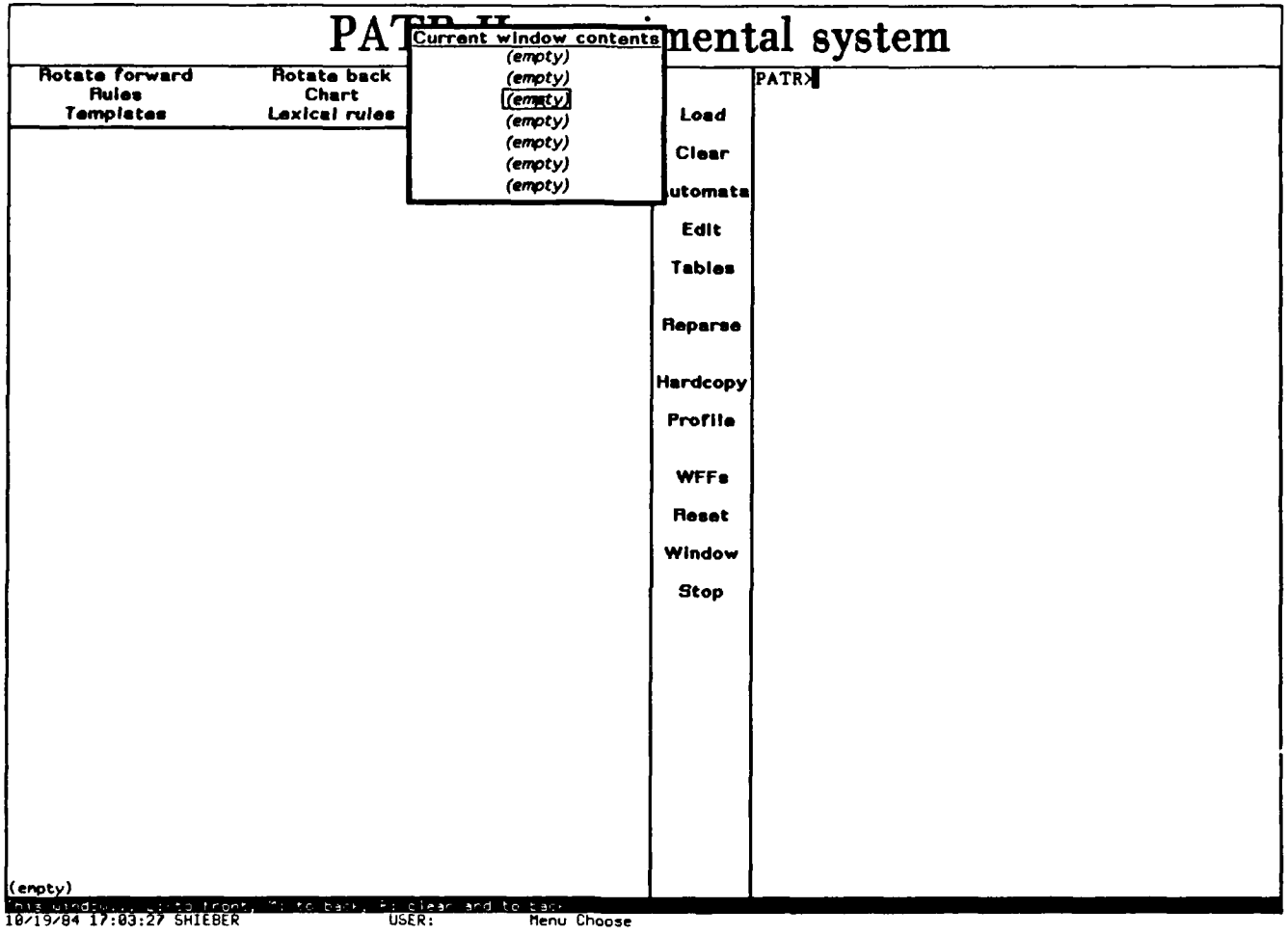
6. The profile men pops up, with information loaded from the user's "klaus.init file" and the user prepares to change the configuration so that active edges are not traced.



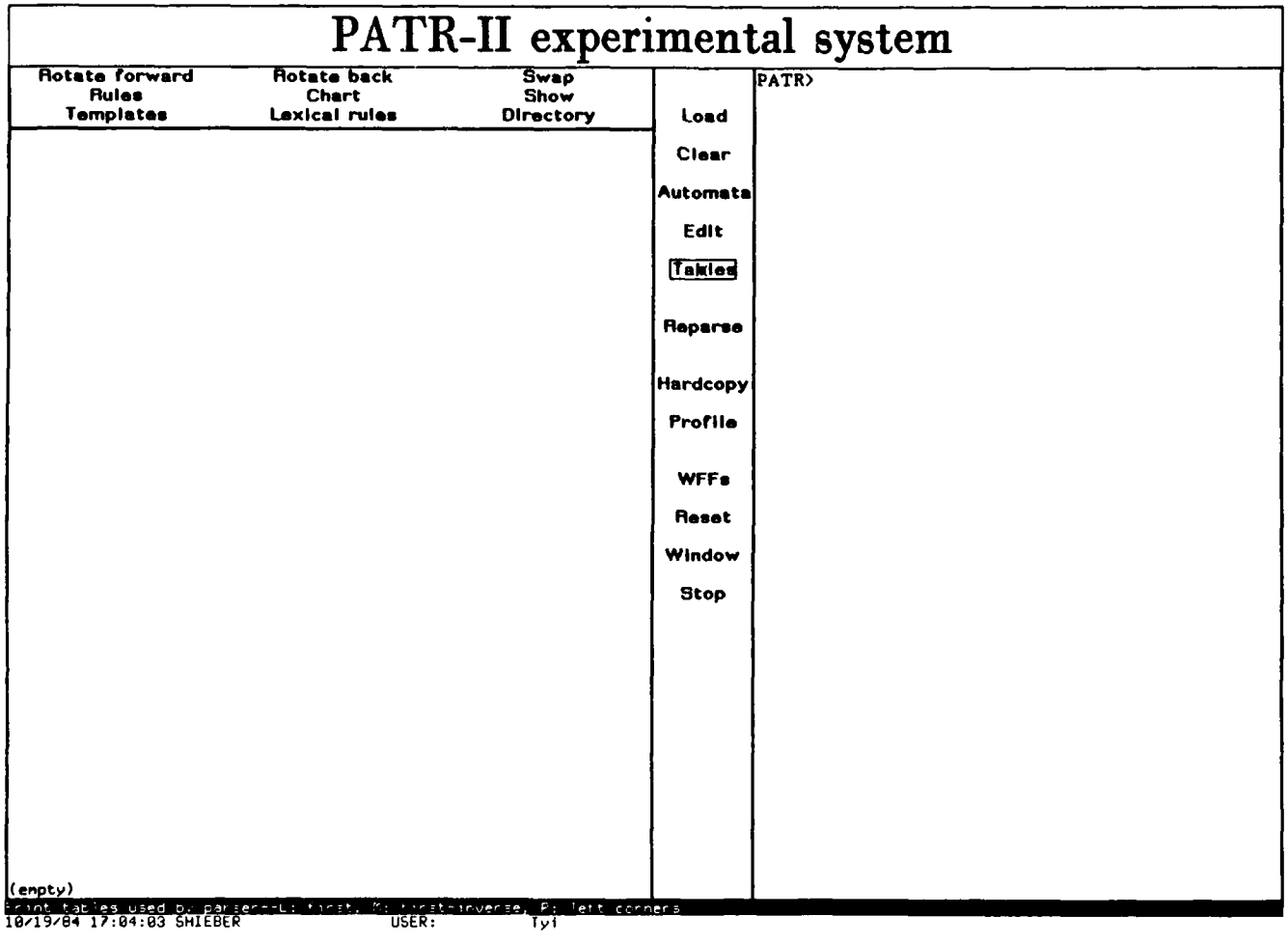
7. The change is made by clicking on the appropriate item. The user exits the profile menu and the changes are made.



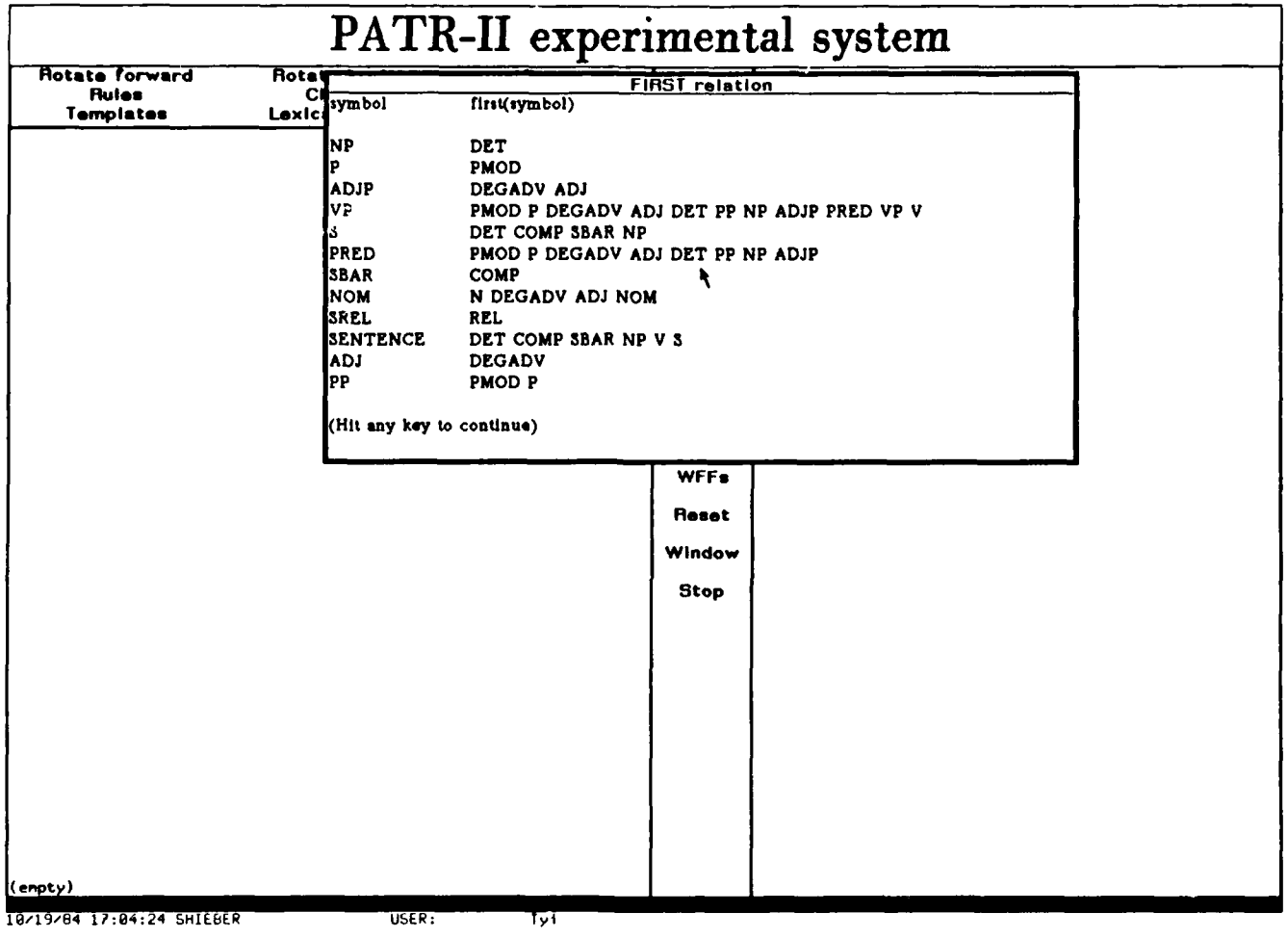
8. The user is about to display the directory showing the contents of the left display windows.



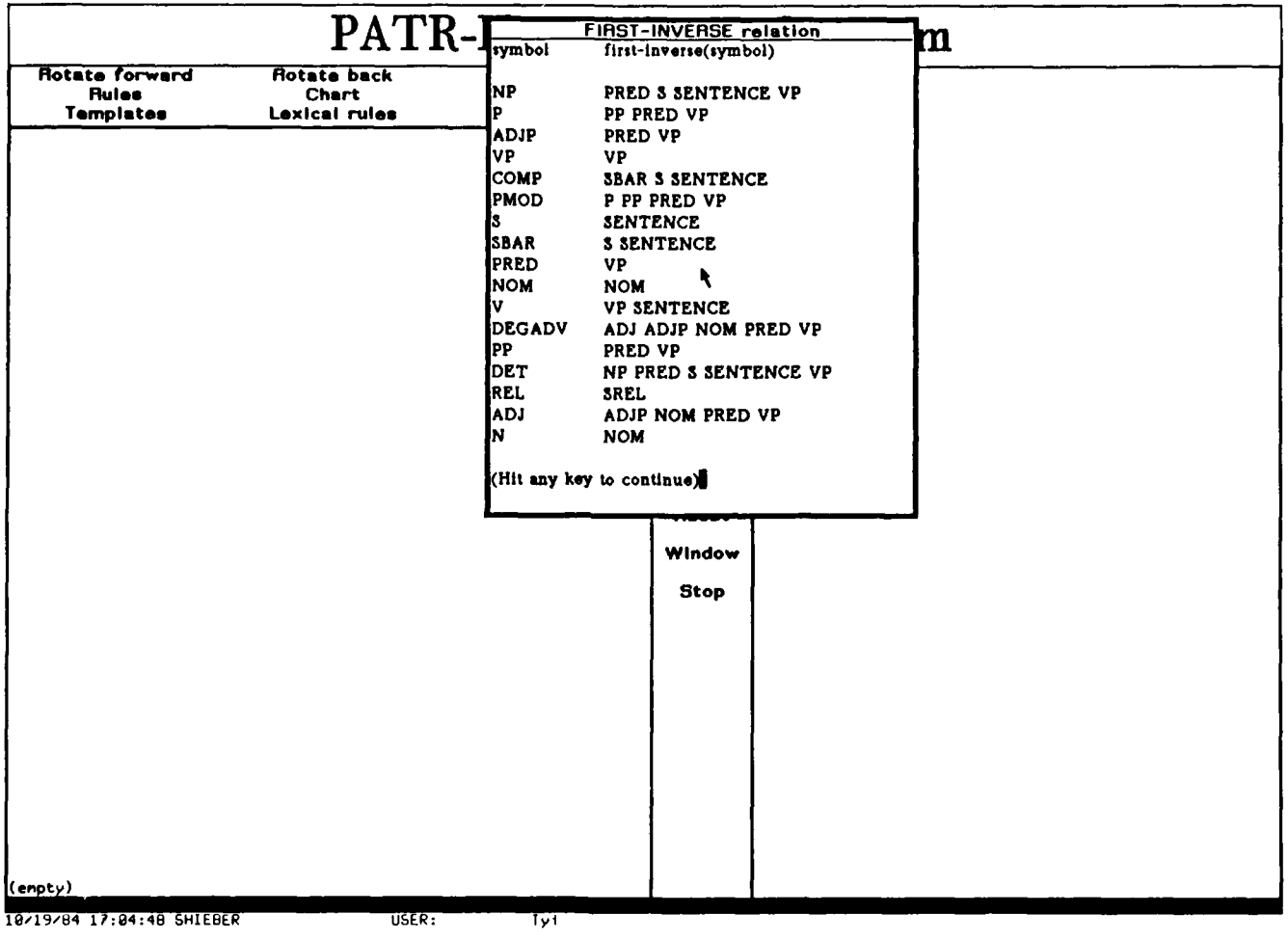
9. They are all empty at this point.



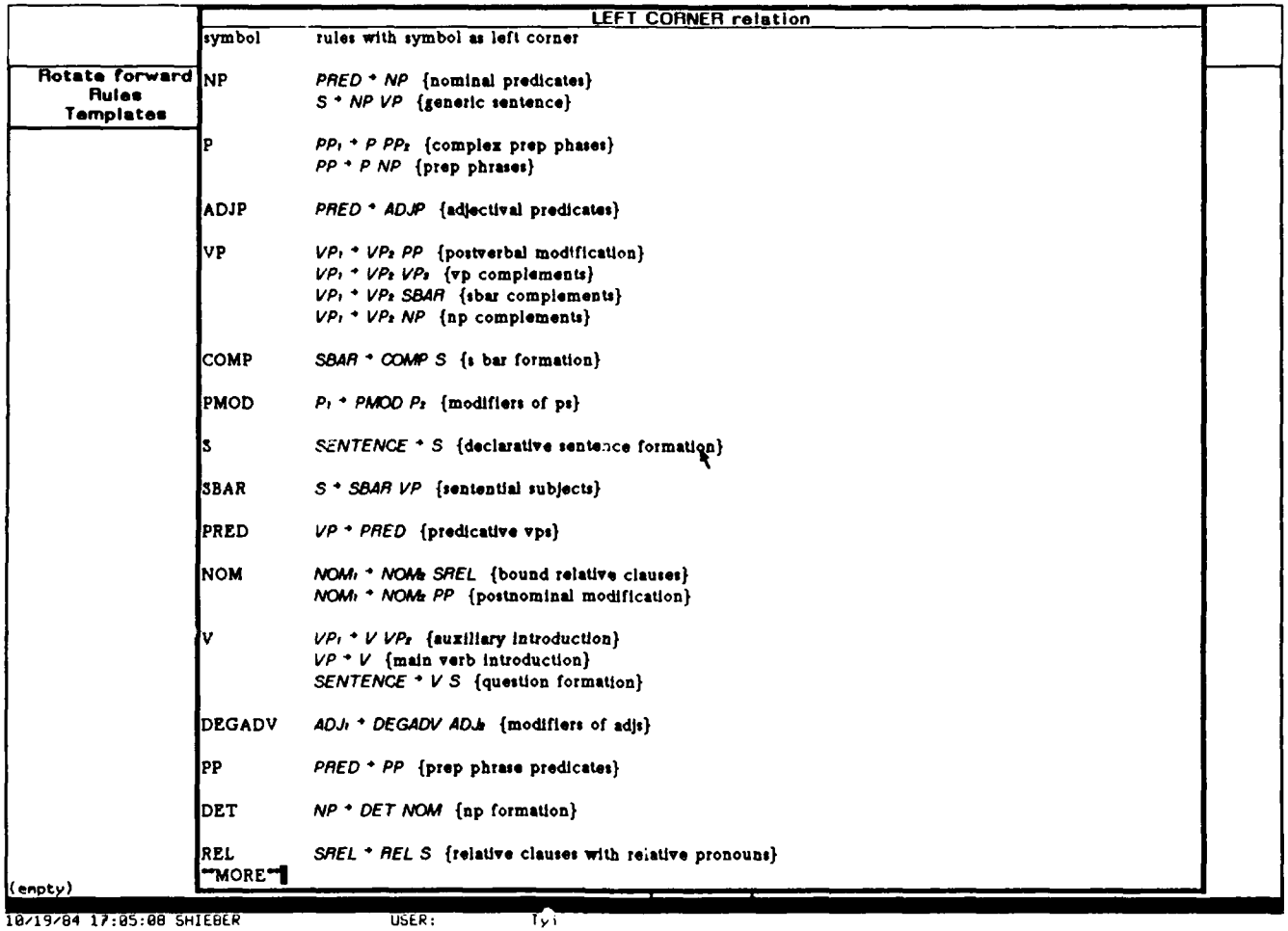
10. The user is about to click left on "Tables" to show the table of the "first" relation.



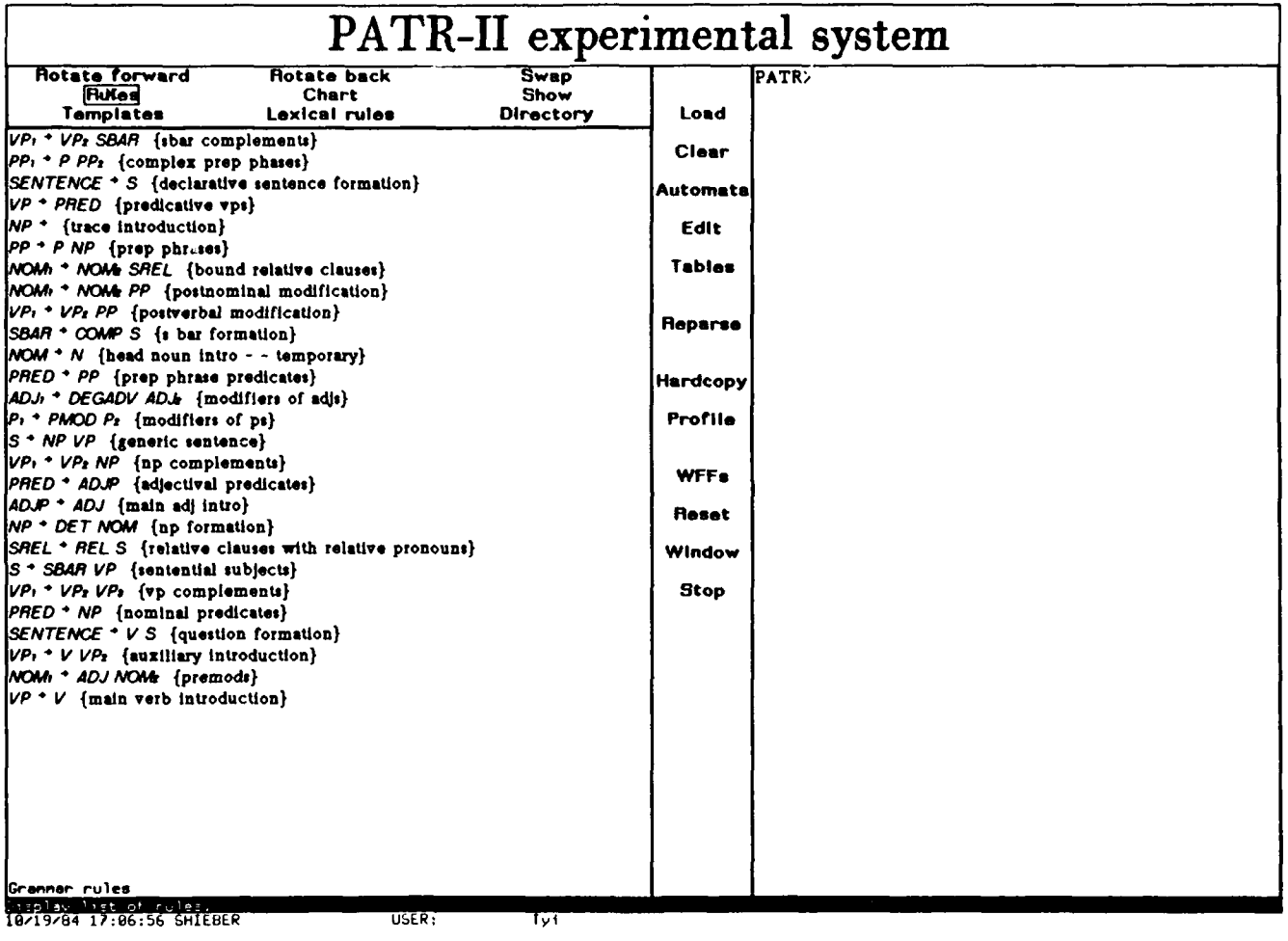
11. The "first" relation is displayed.



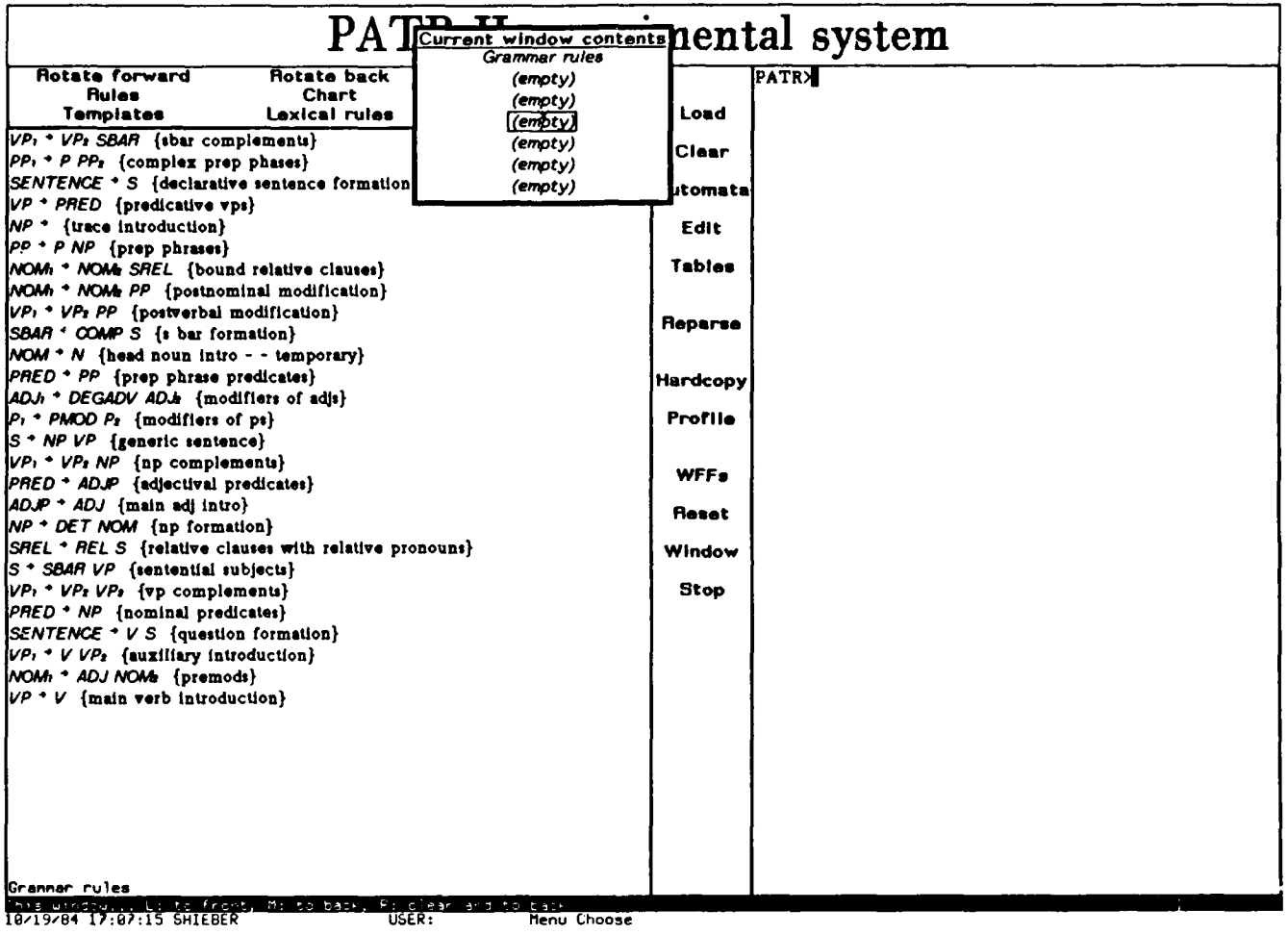
12. Clicking middle displays the "first-inverse" relation.



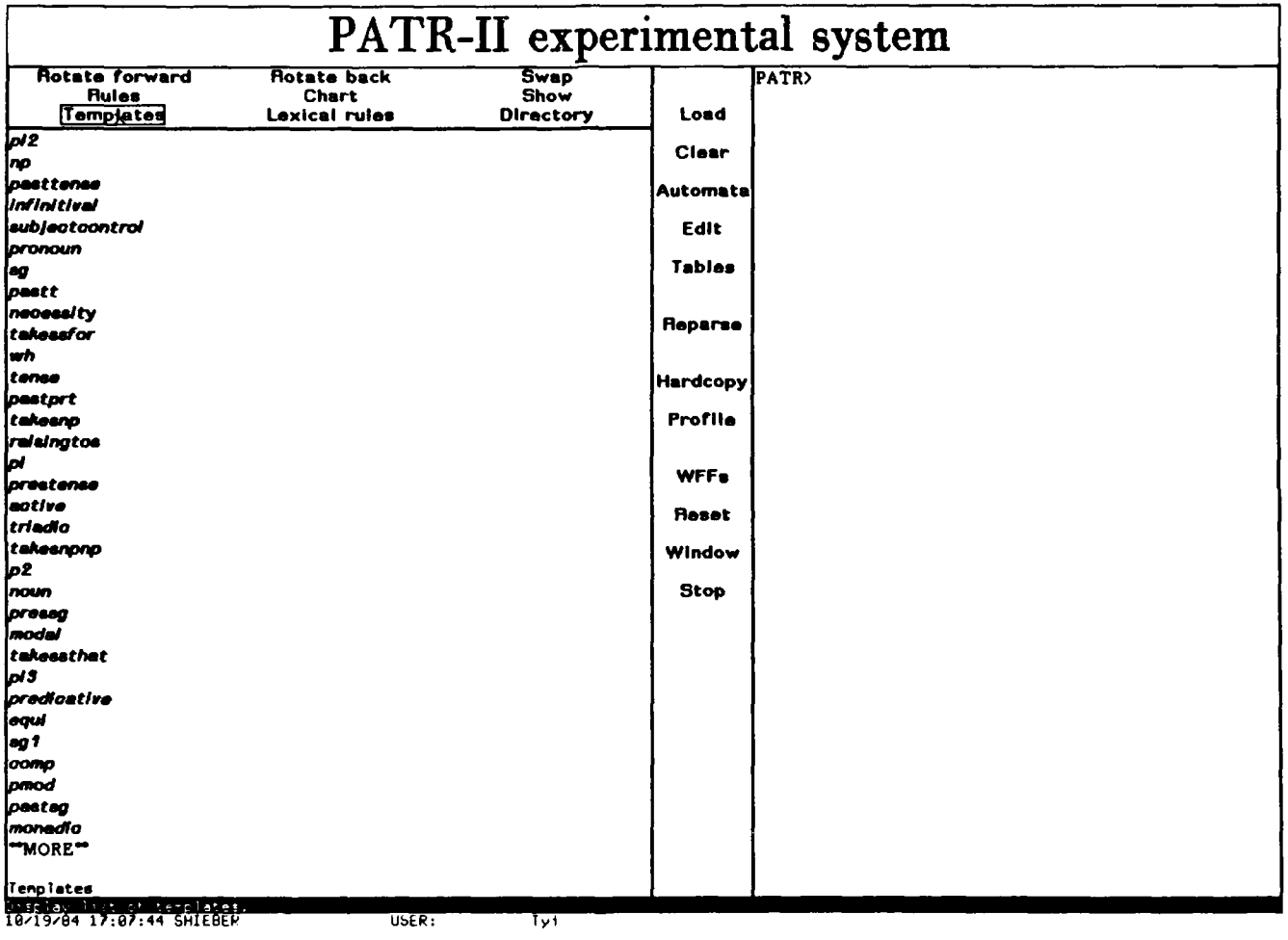
13. Clicking right displays the "left-corner" relation.



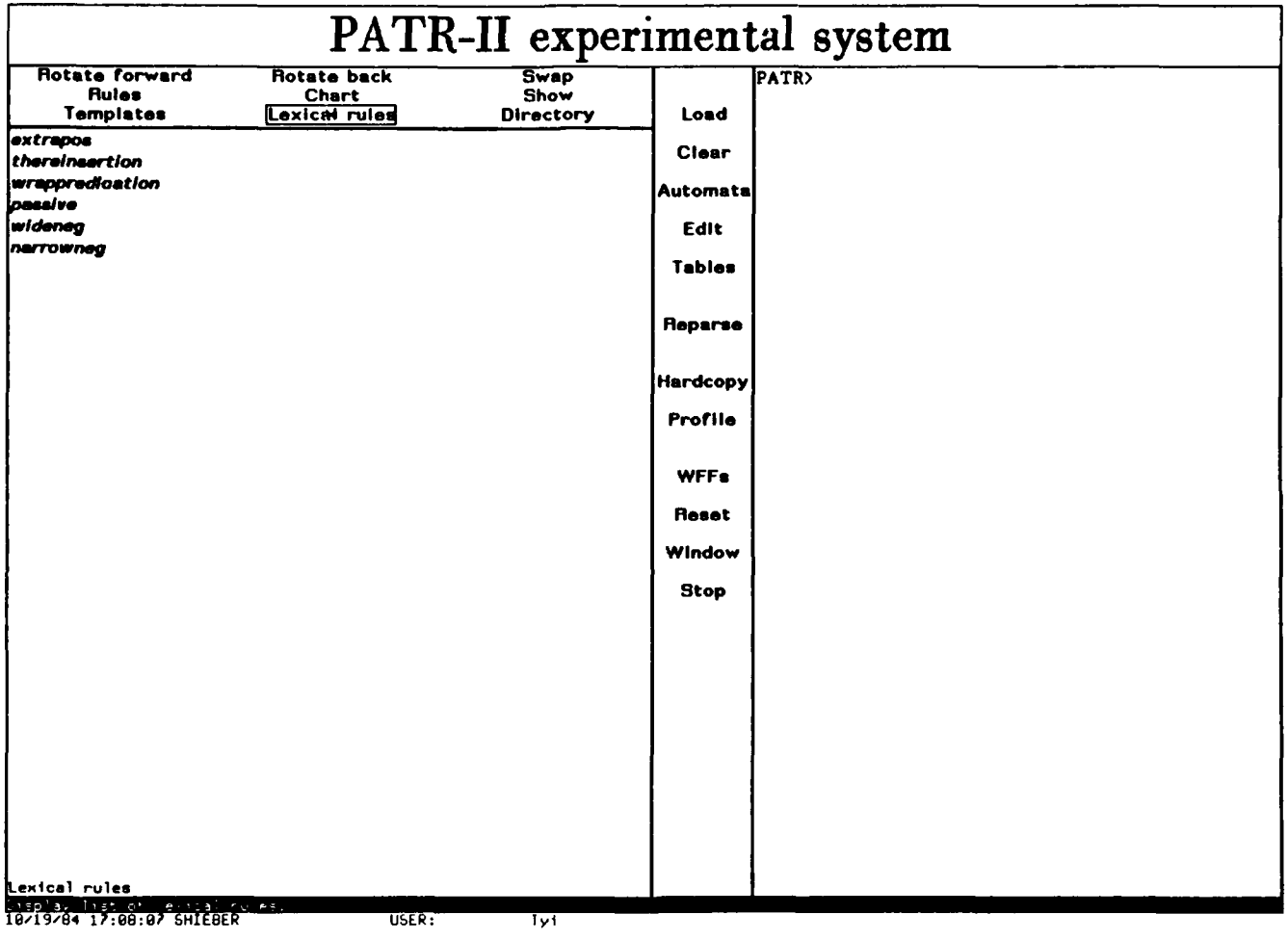
14. The user clicks the "Rules" command in the upper left menu causing a list of grammar rules to be displayed in an available window.



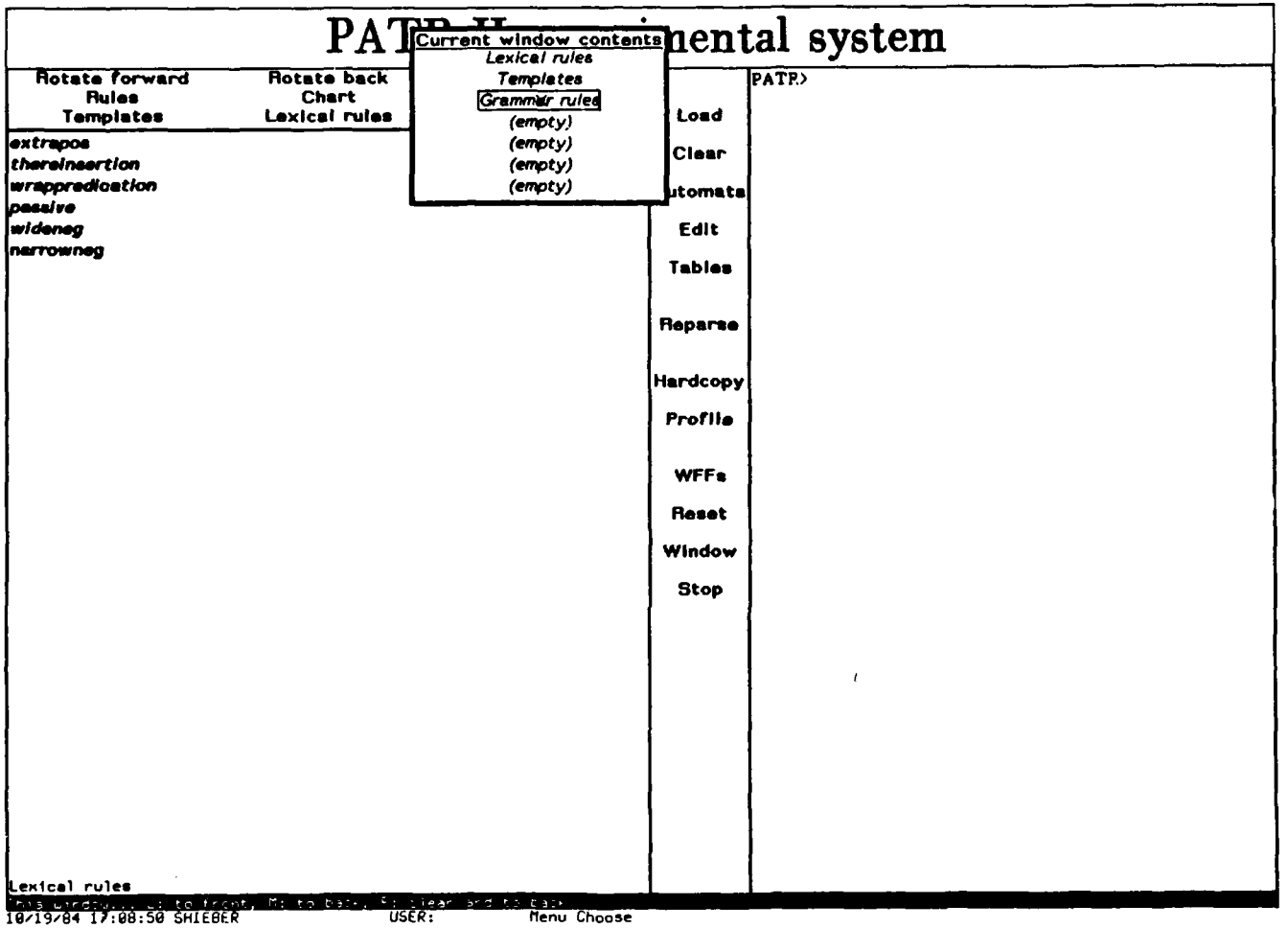
15. Now displaying the directory indicates that the top window holds a list of grammar rules.



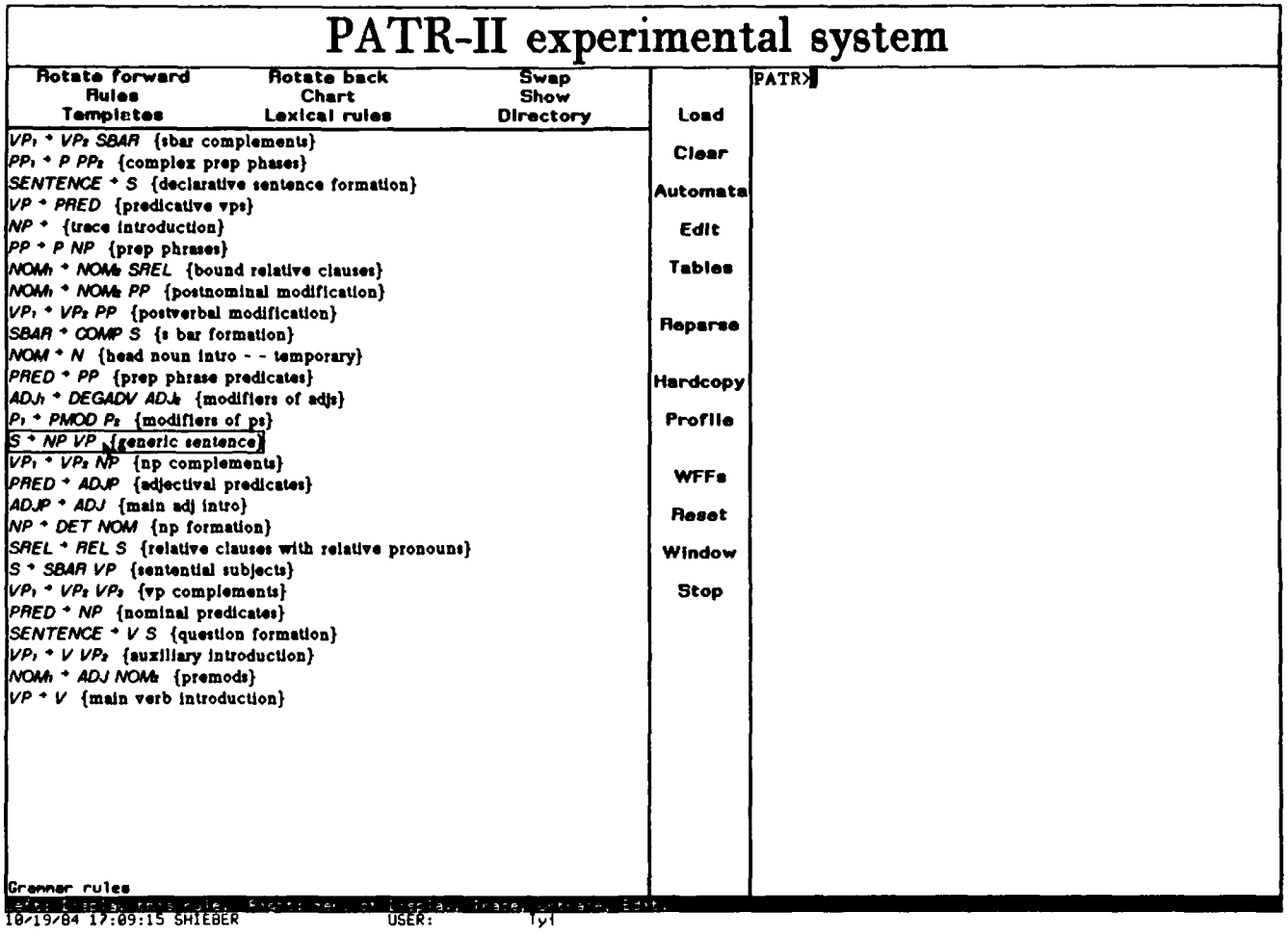
16. Clicking the "Templates" command displays a list of templates.



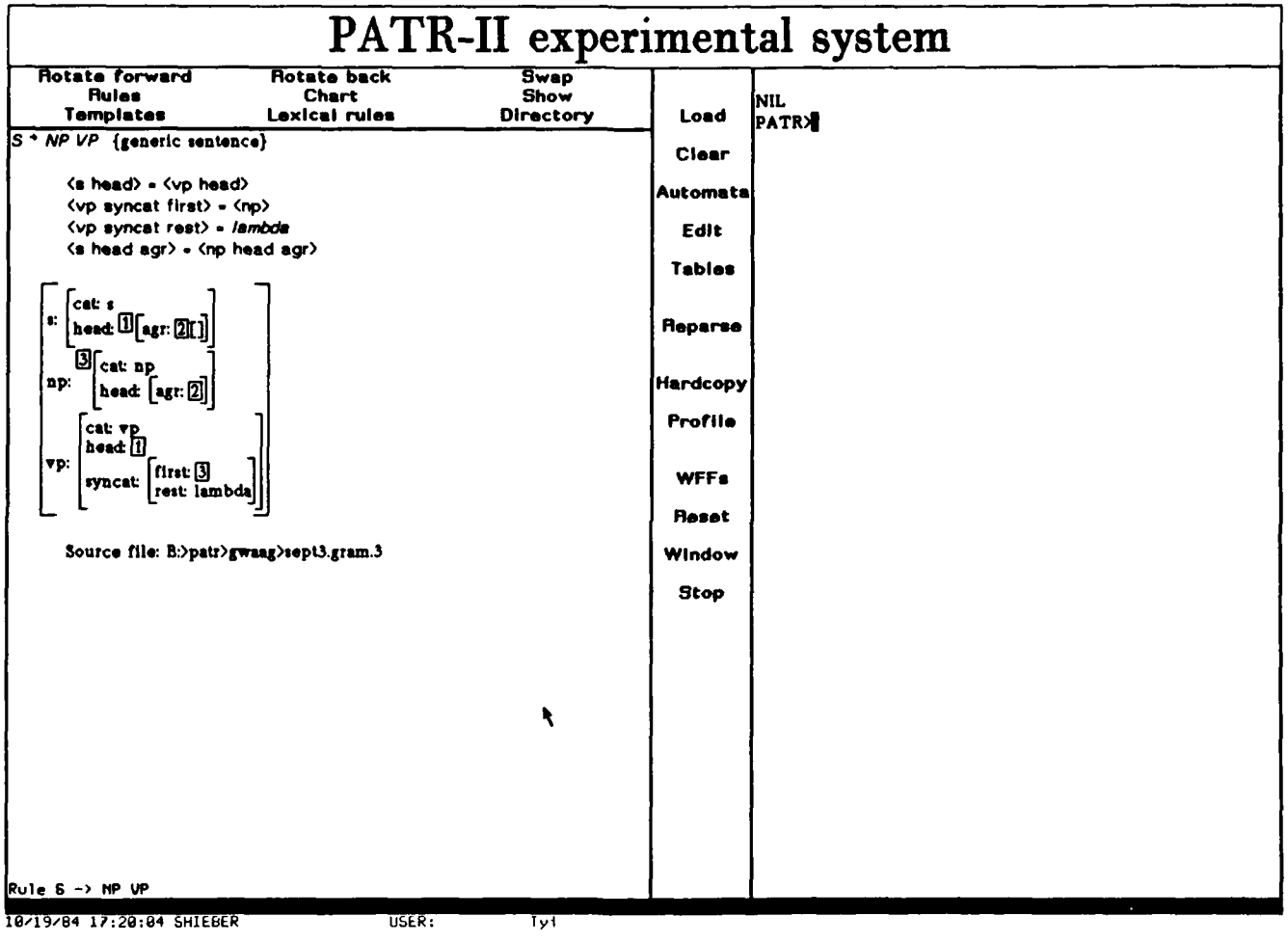
17. Clicking the "Lexical rules" command displays a list of lexical rules.



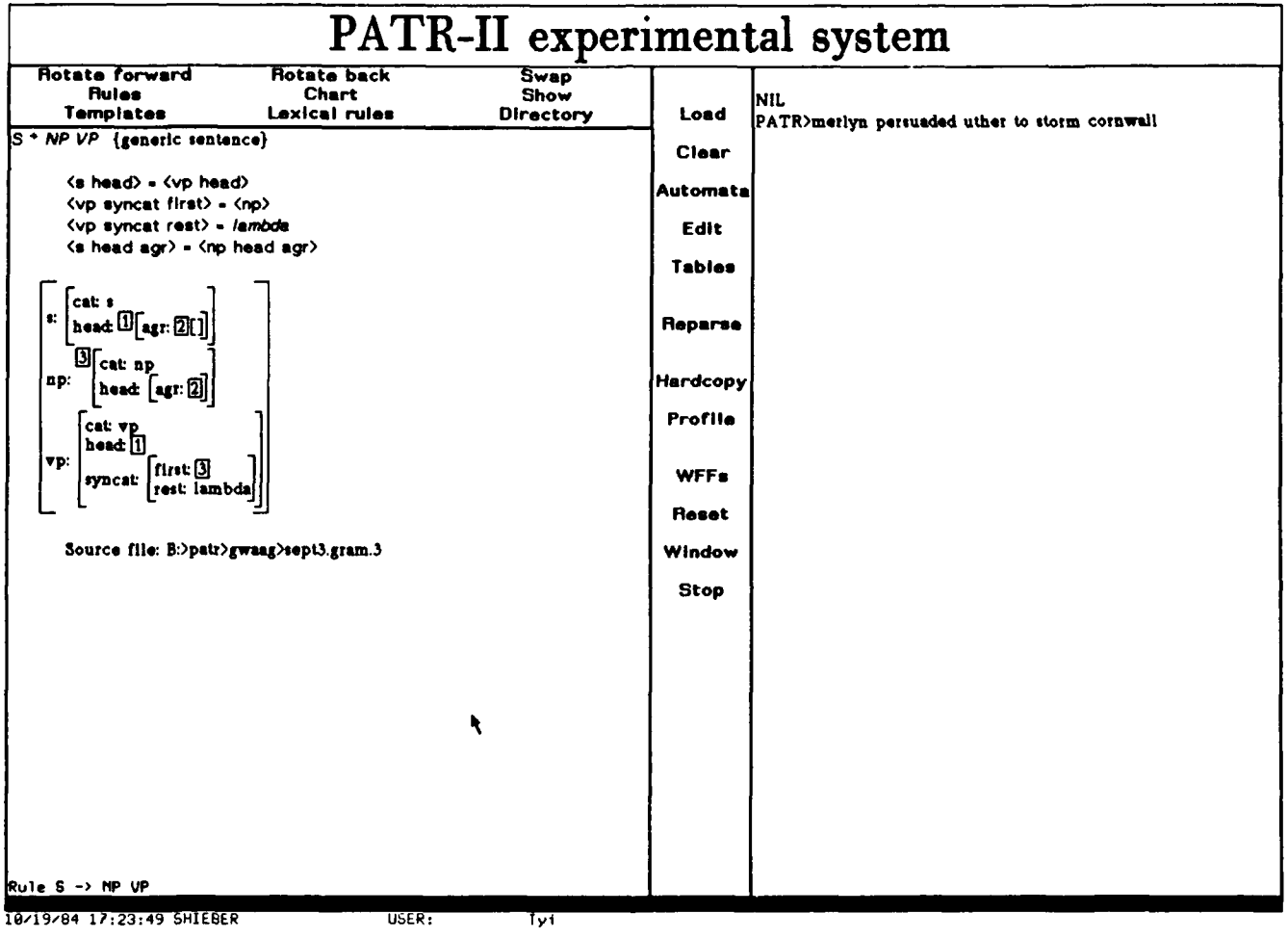
18. The directory now indicates that three windows have information in them. The order, from top to bottom, corresponds to recency, the lexical rules being the most recently exposed. The user is about to click on the third entry in the directory, thereby moving the corresponding window to the top of the stack and exposing it.



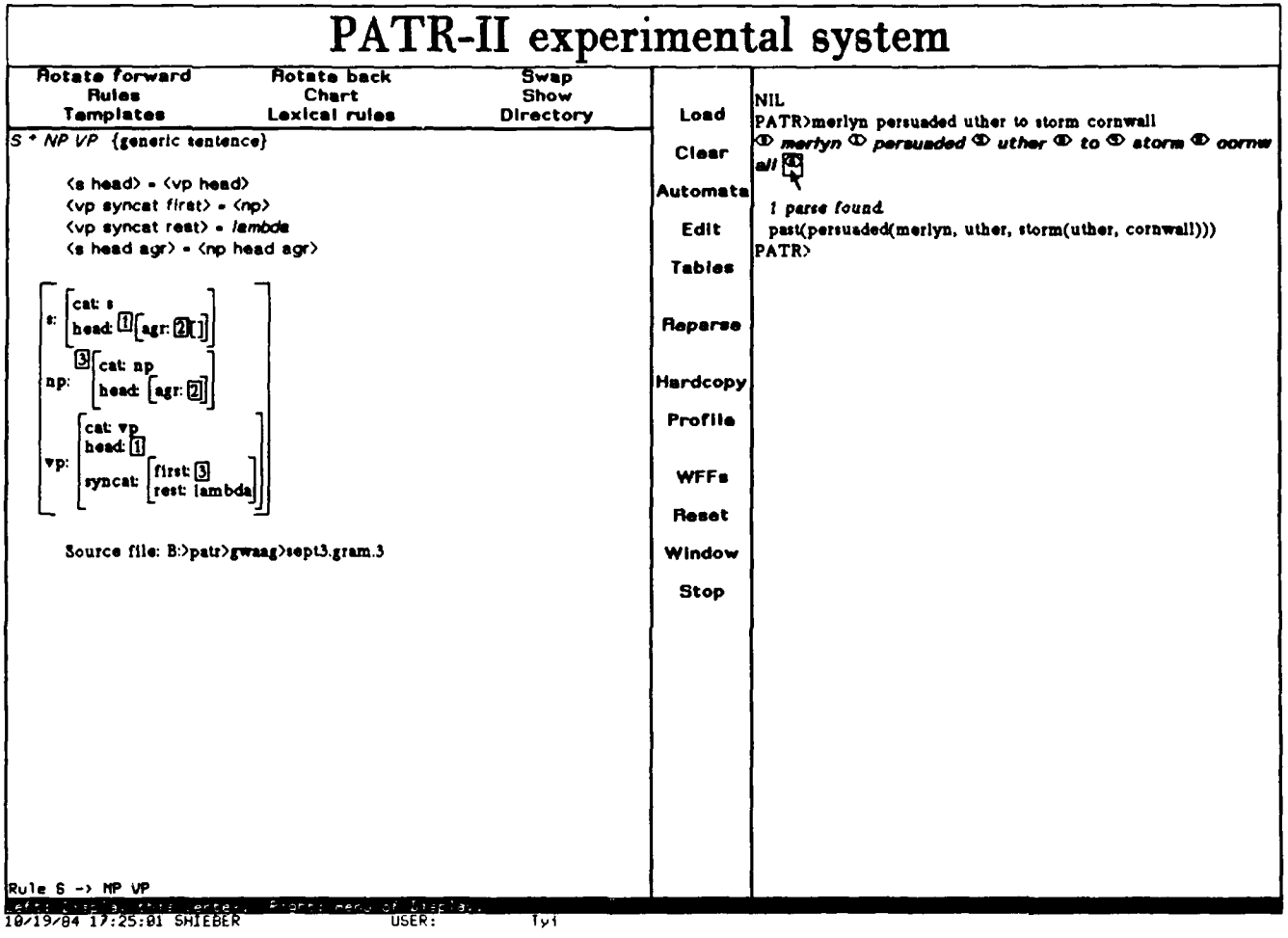
19. The grammar rules thus come into view. The user wants to display the "generic sentence" rule, so he clicks left on it.



20. The rule is displayed.

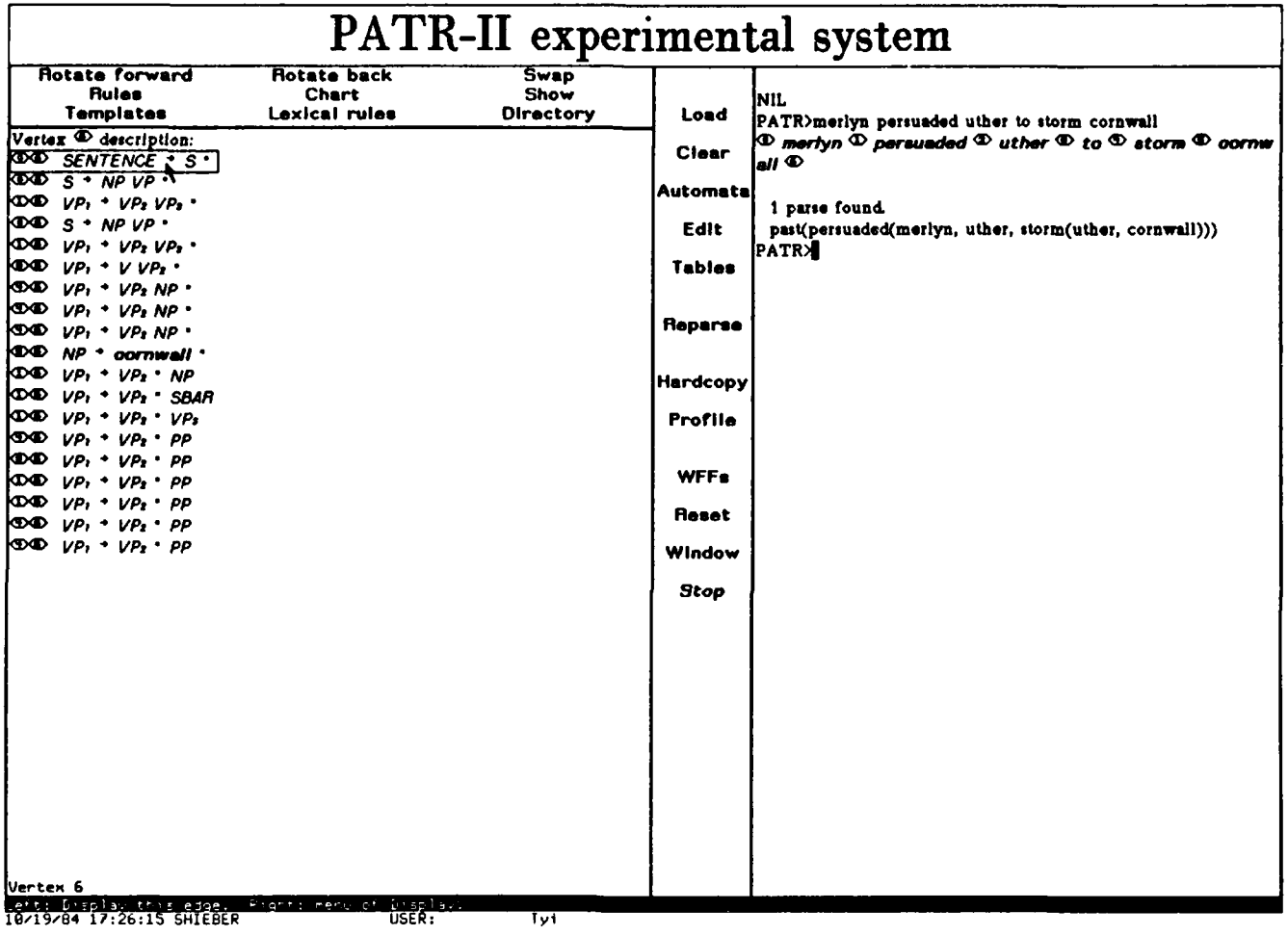


21. The user types in a sentence to have it parsed by the system.



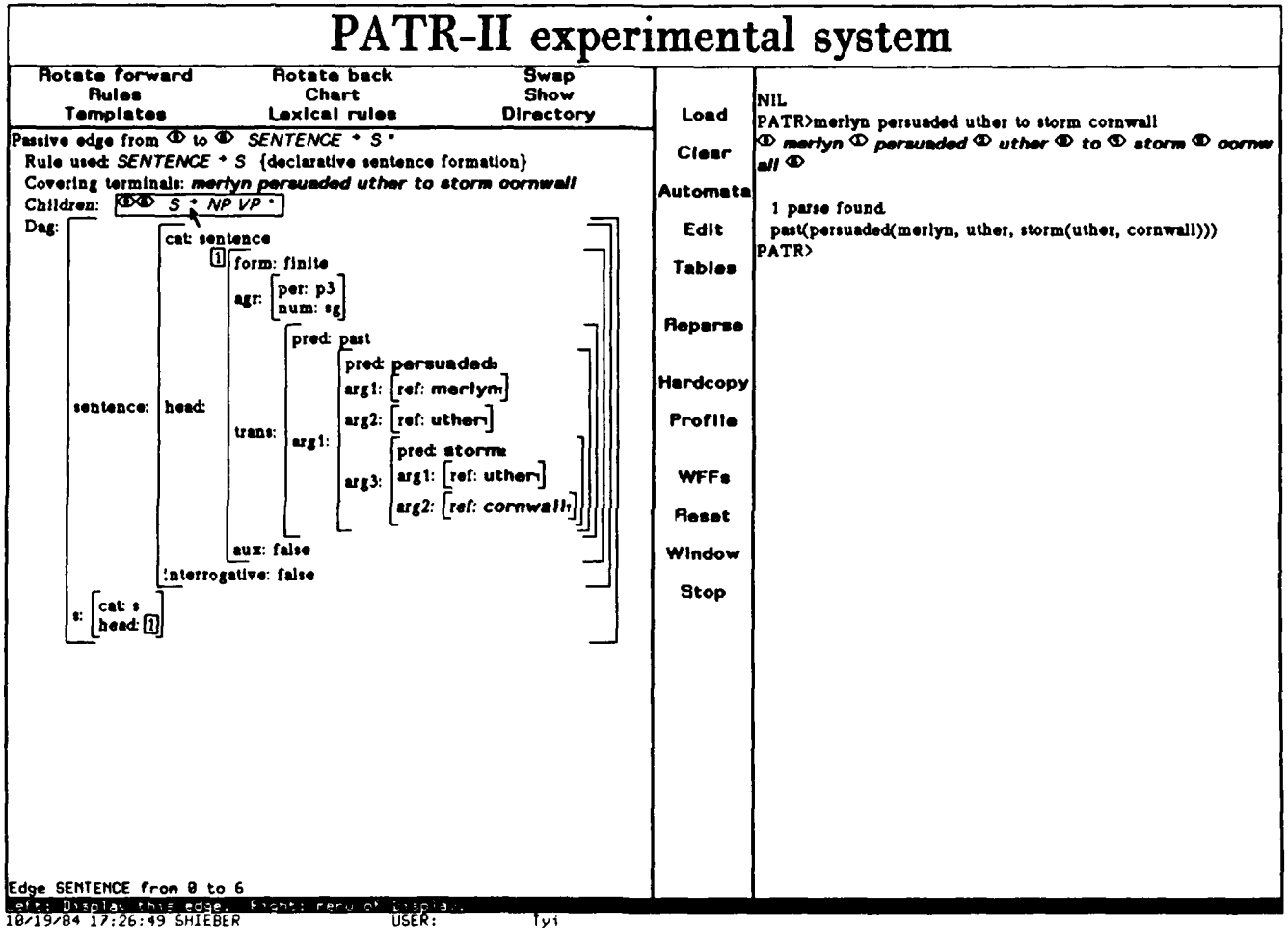
22. The system finds a single parse for the sentence, and prints out a representation of the translation of the sentence into a logical form language.

    The user wants to see the edges coming into the final vertex in the chart, so he clicks on an icon of that vertex.

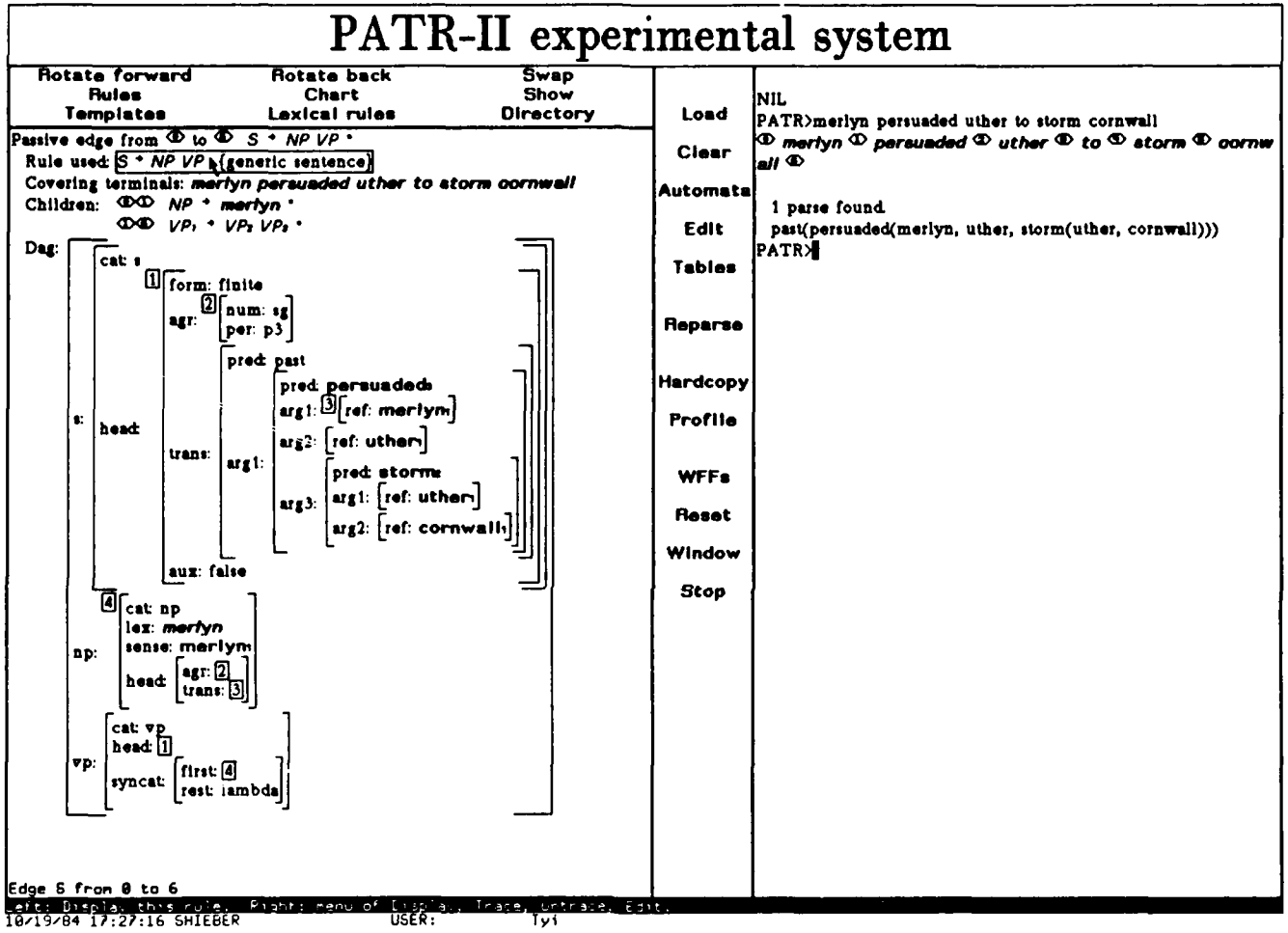


23. The vertex information is displayed in an available display window.

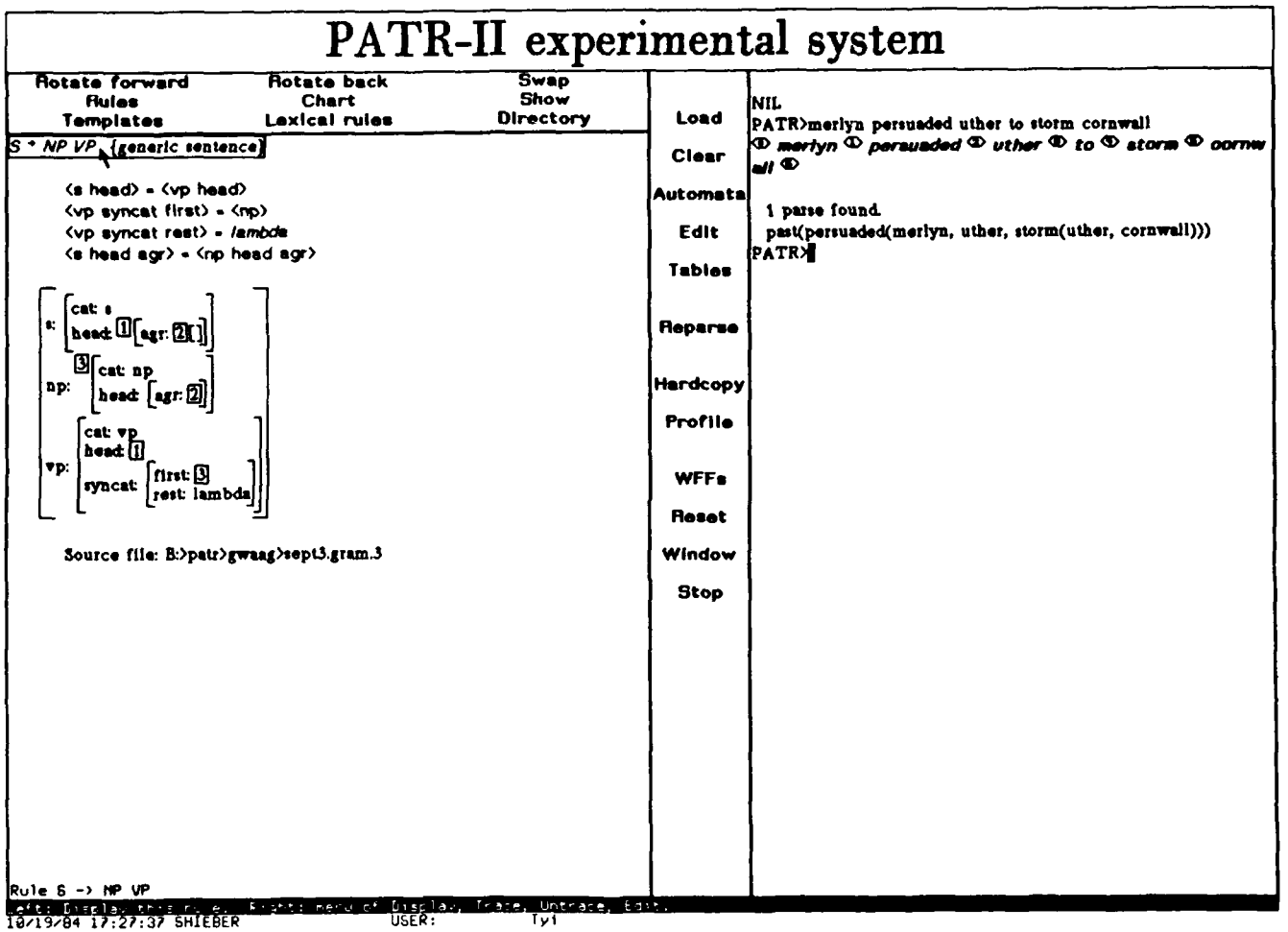
The user picks an edge to display and clicks on it.



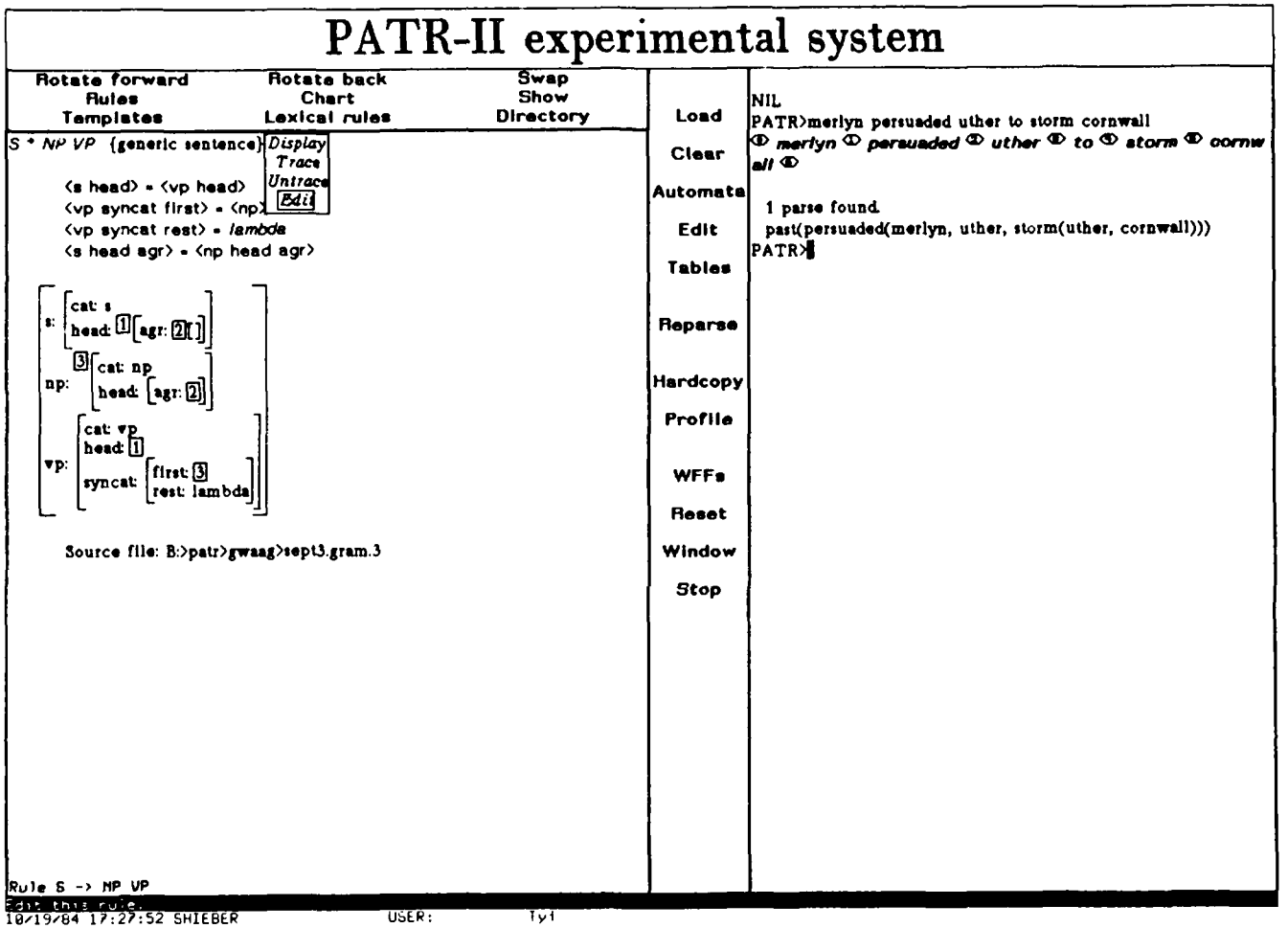
24. The edge is displayed. The user clicks on the child constituent of this edge, to display it.



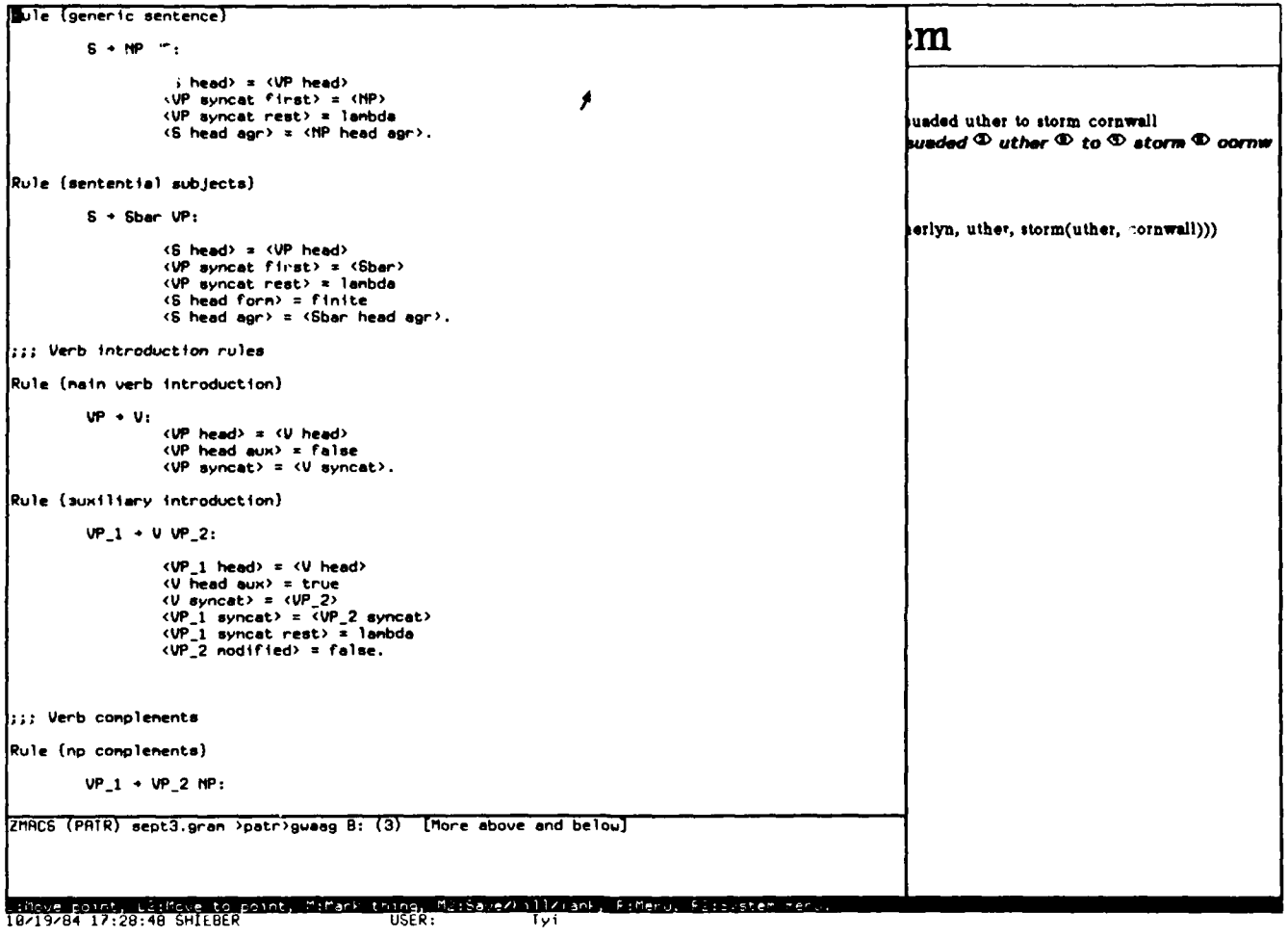
25. It in turn is displayed. The user clicks on the rule that was used in forming this edge, the "generic sentence" rule.



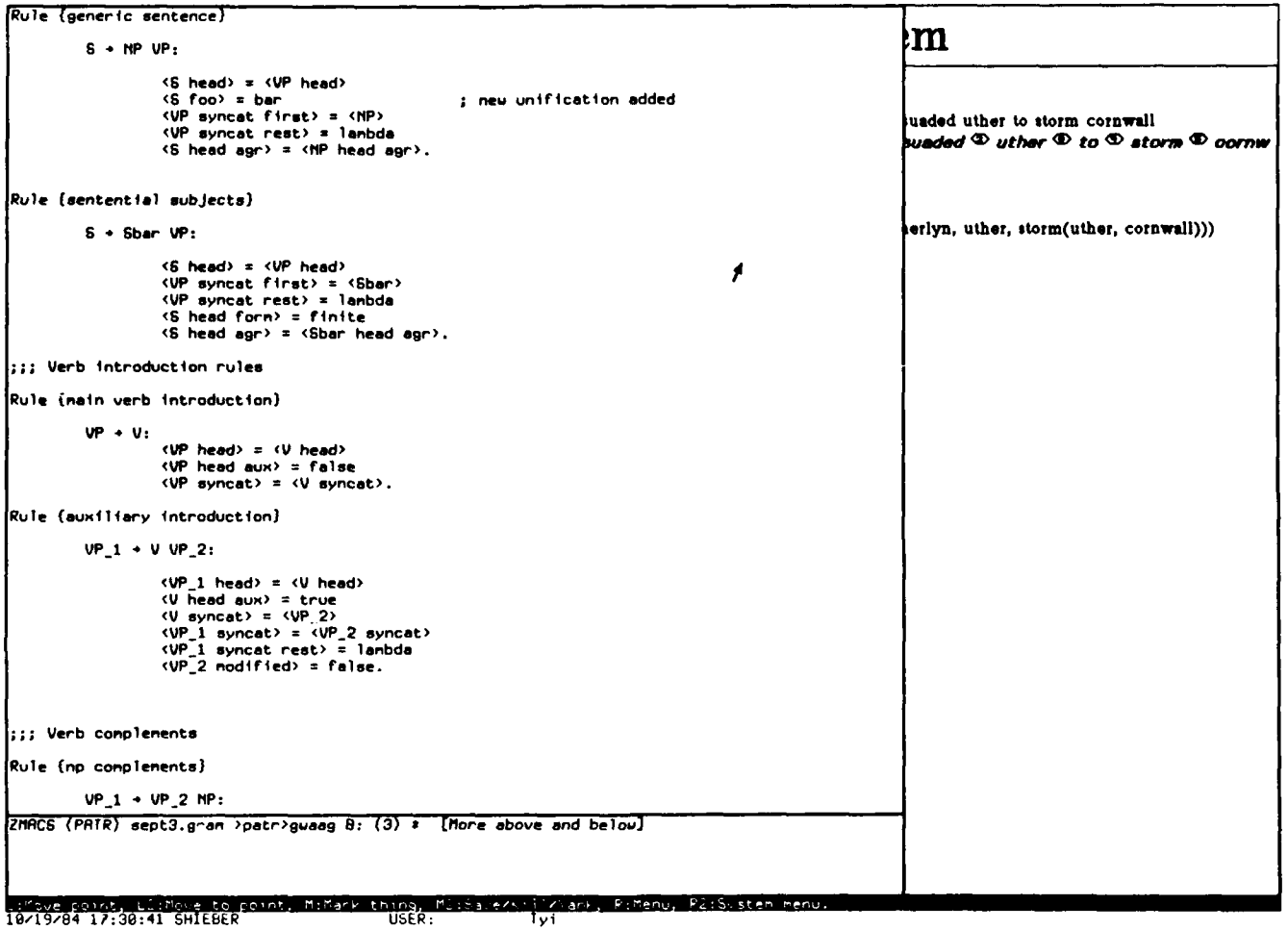
26. The rule is displayed again. The user wants to edit this rule, so he clicks right on the icon for the rule.



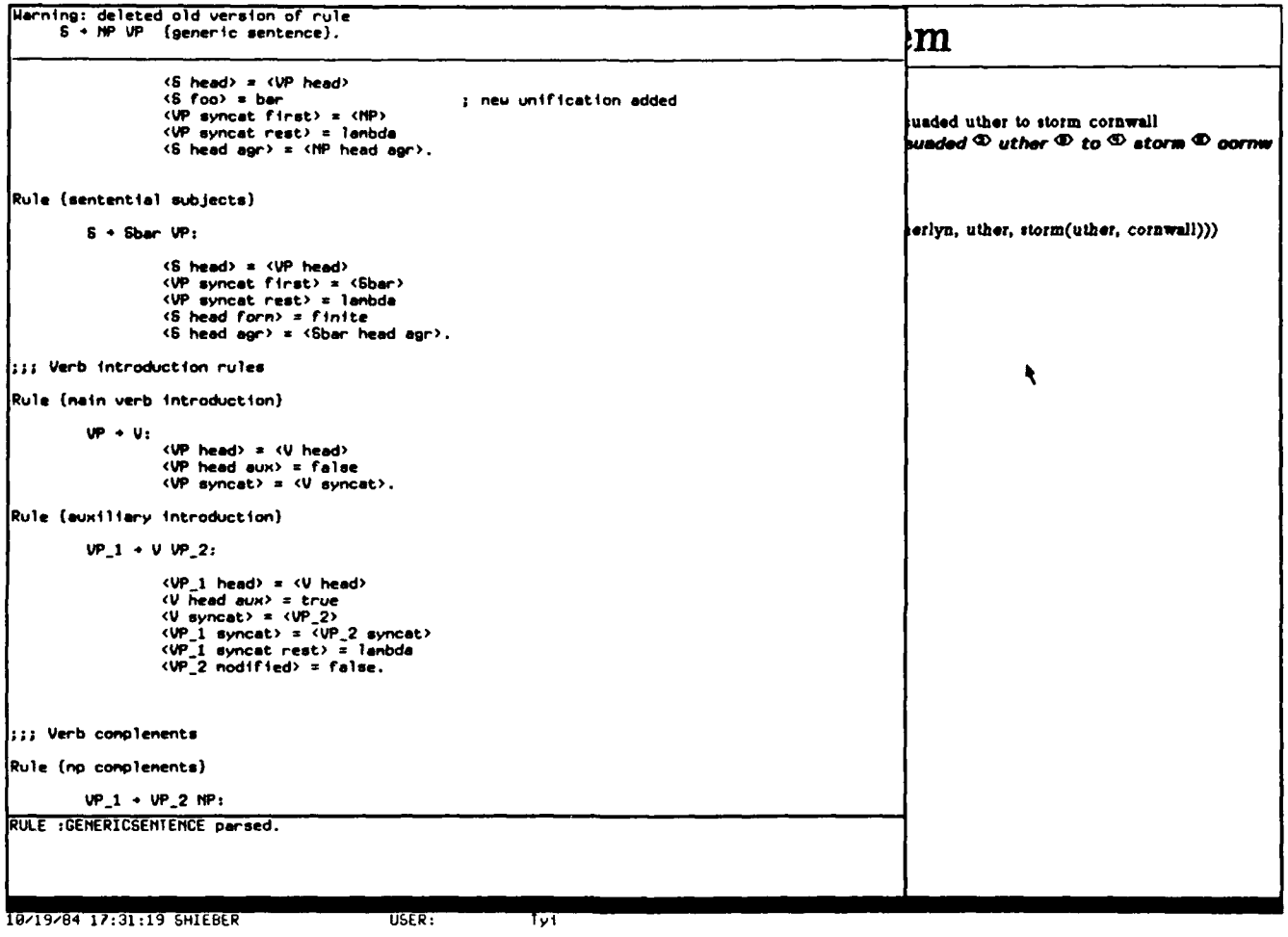
27. A menu pops up allowing him to choose an operation to perform on this rule. He picks the "Edit" option.



28. The system pulls up the ZMACS editor window, loads the source file for the rule, and positions the cursor at the beginning of the rule definition, as seen in this snapshot.

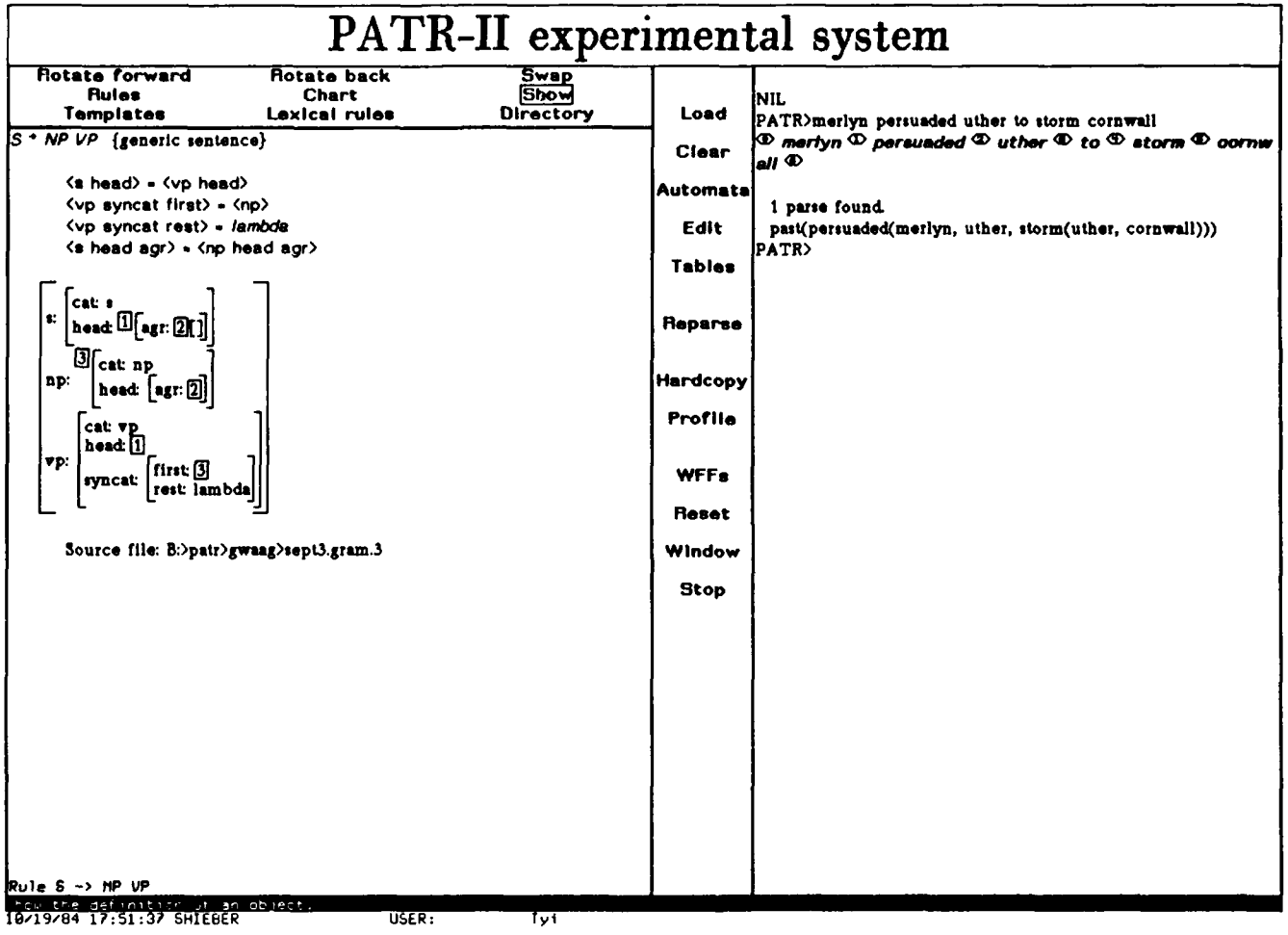


29. The user adds a unification to the rule using normal ZMACS editing commands.

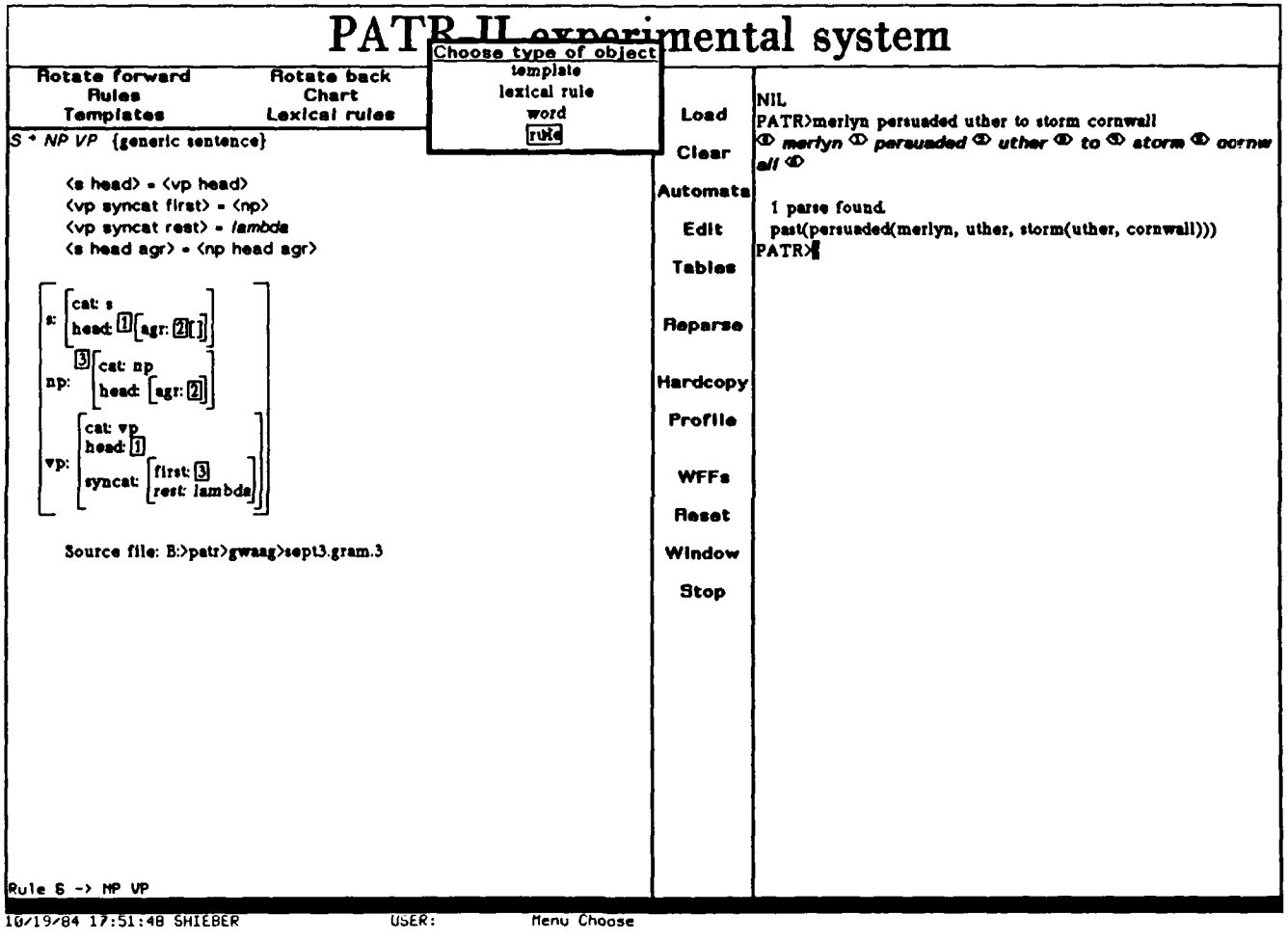


30. The ZMACS command control-shift-C causes the rule to be incrementally compiled into the system. A warning is given that it is replacing the old version of the rule with the new version.

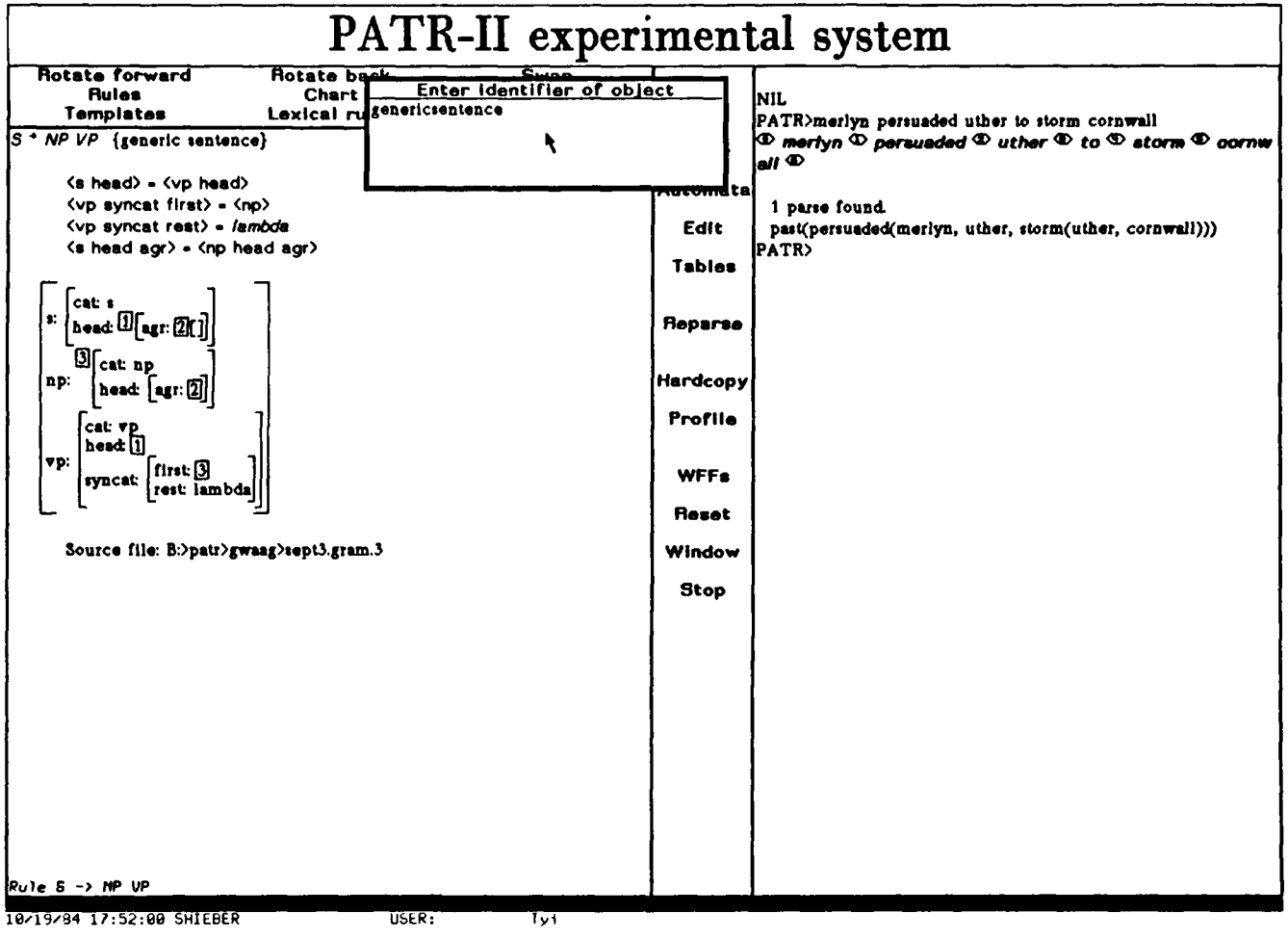
    The user is about to click the mouse while the mouse cursor is over the PATR-II window, not the editor window. This will reexpose the PATR-II configuration.



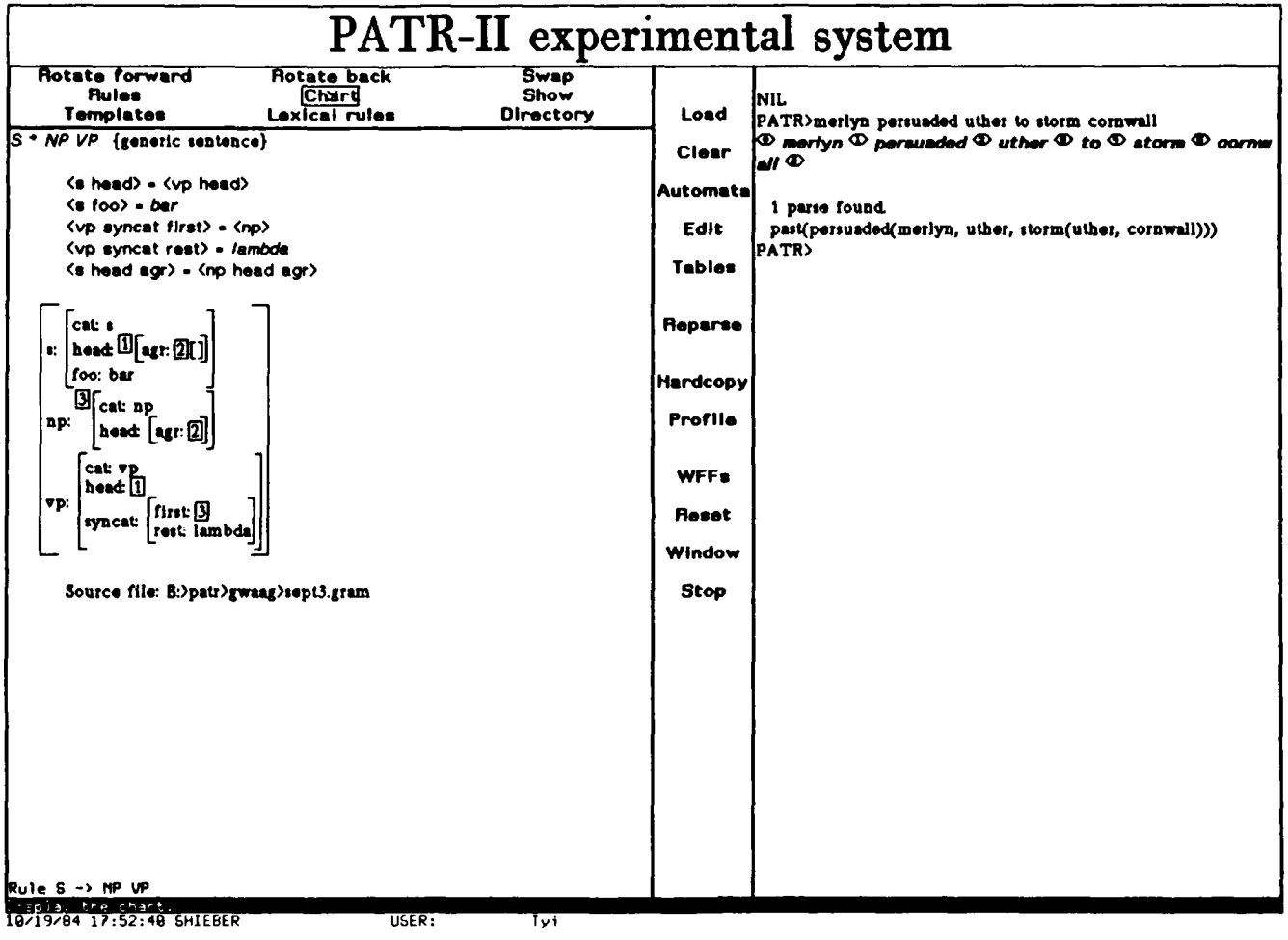
31. The PATR-II window is reexposed. The user wants to see the new version of the "generic sentence" rule, so clicks on "Show".



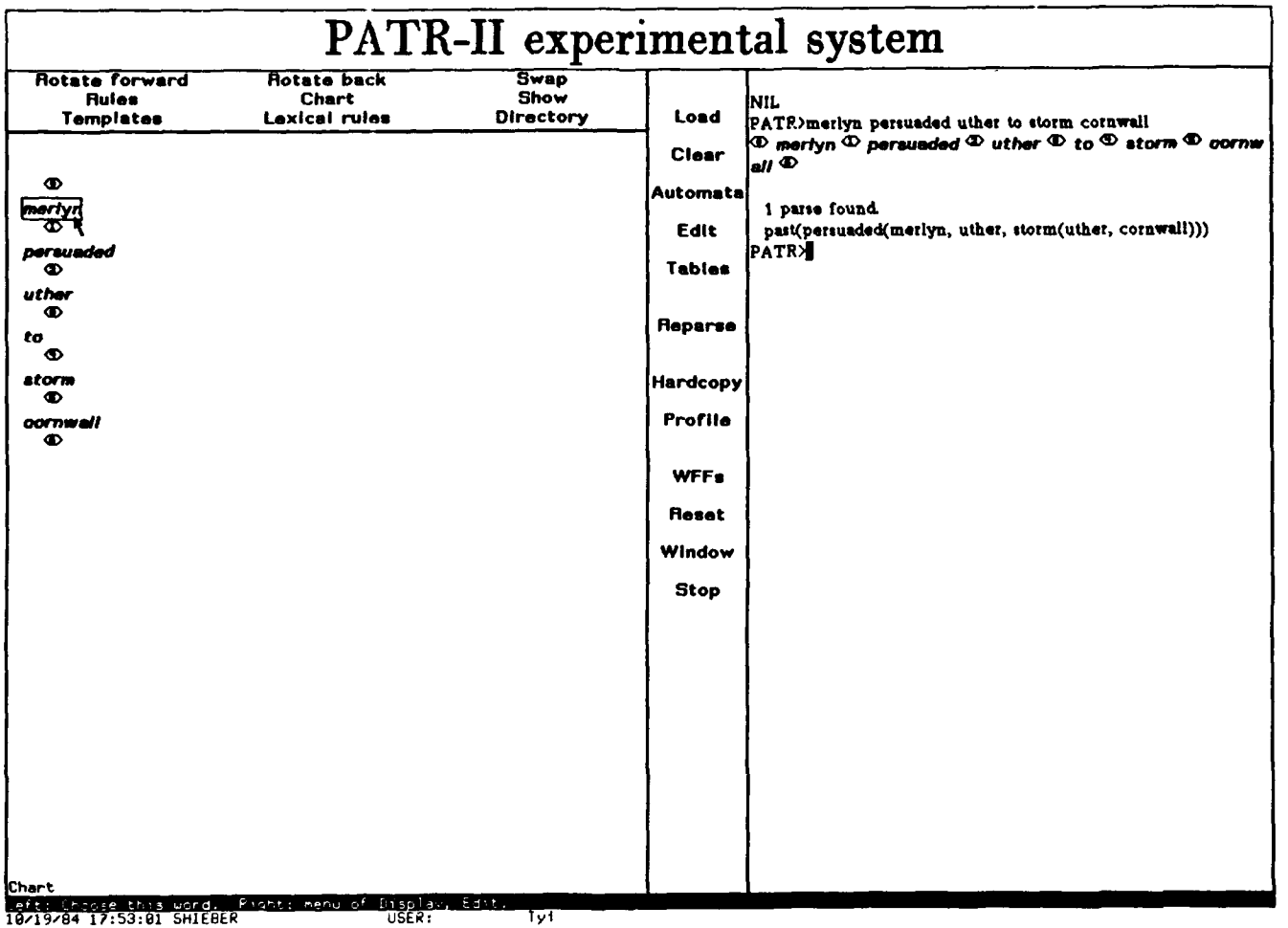
32. From the pop up menu, he chooses "Rule".



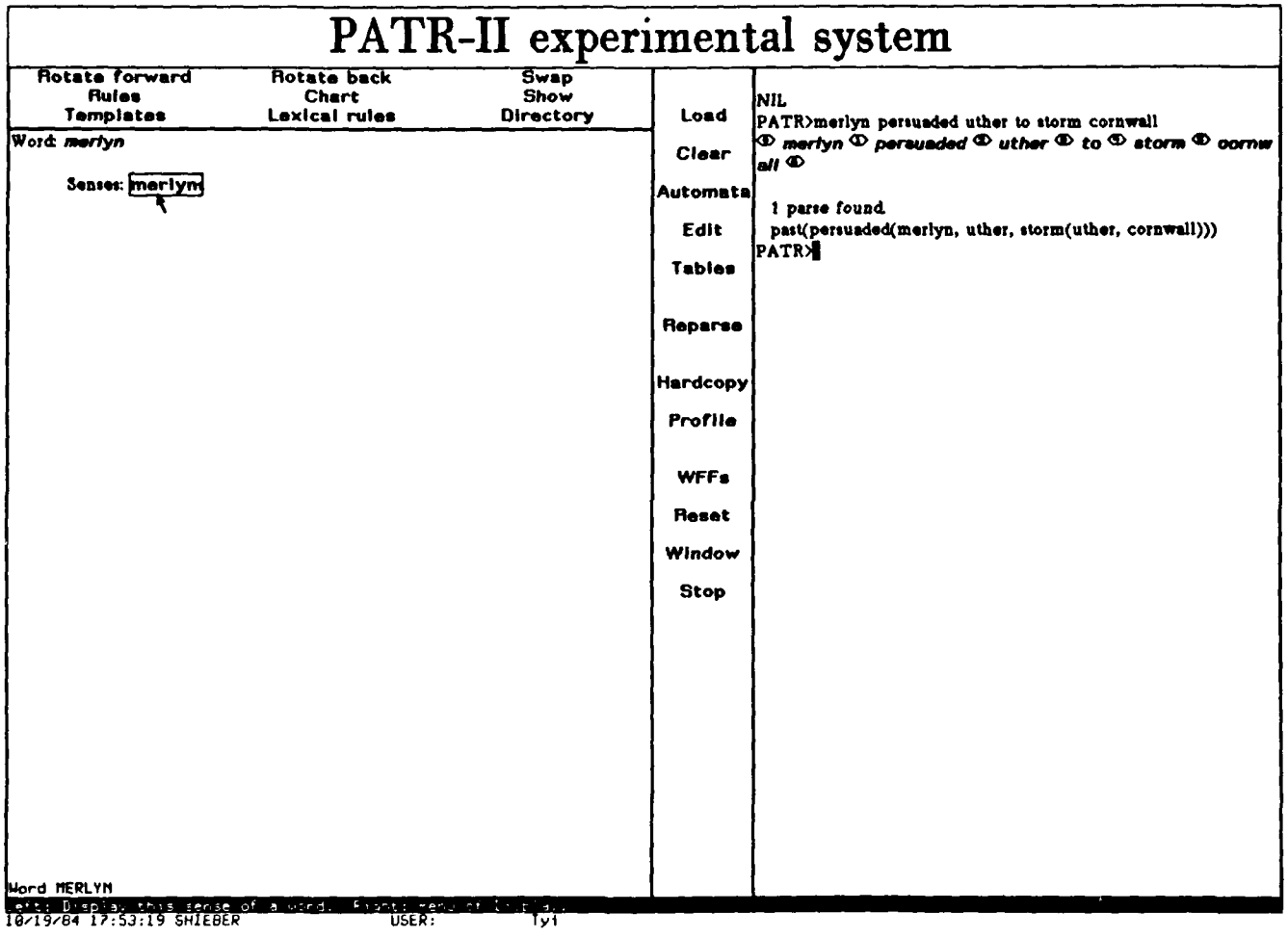
33. The user enters the identifier for the rule he wishes to see.



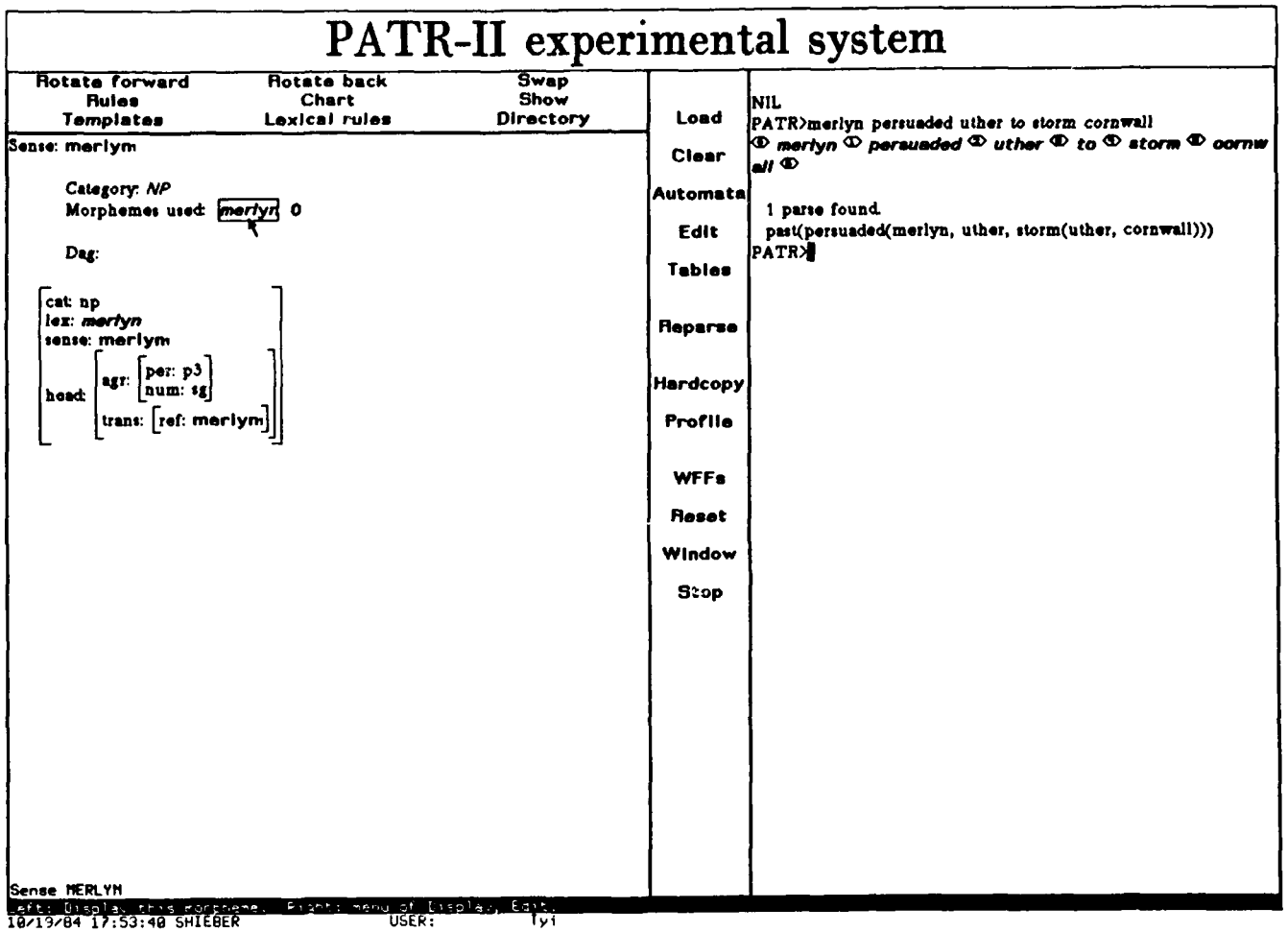
34. The rule is displayed. The user wants to examine the chart (although he could as well use the chart icons appearing in the right window).



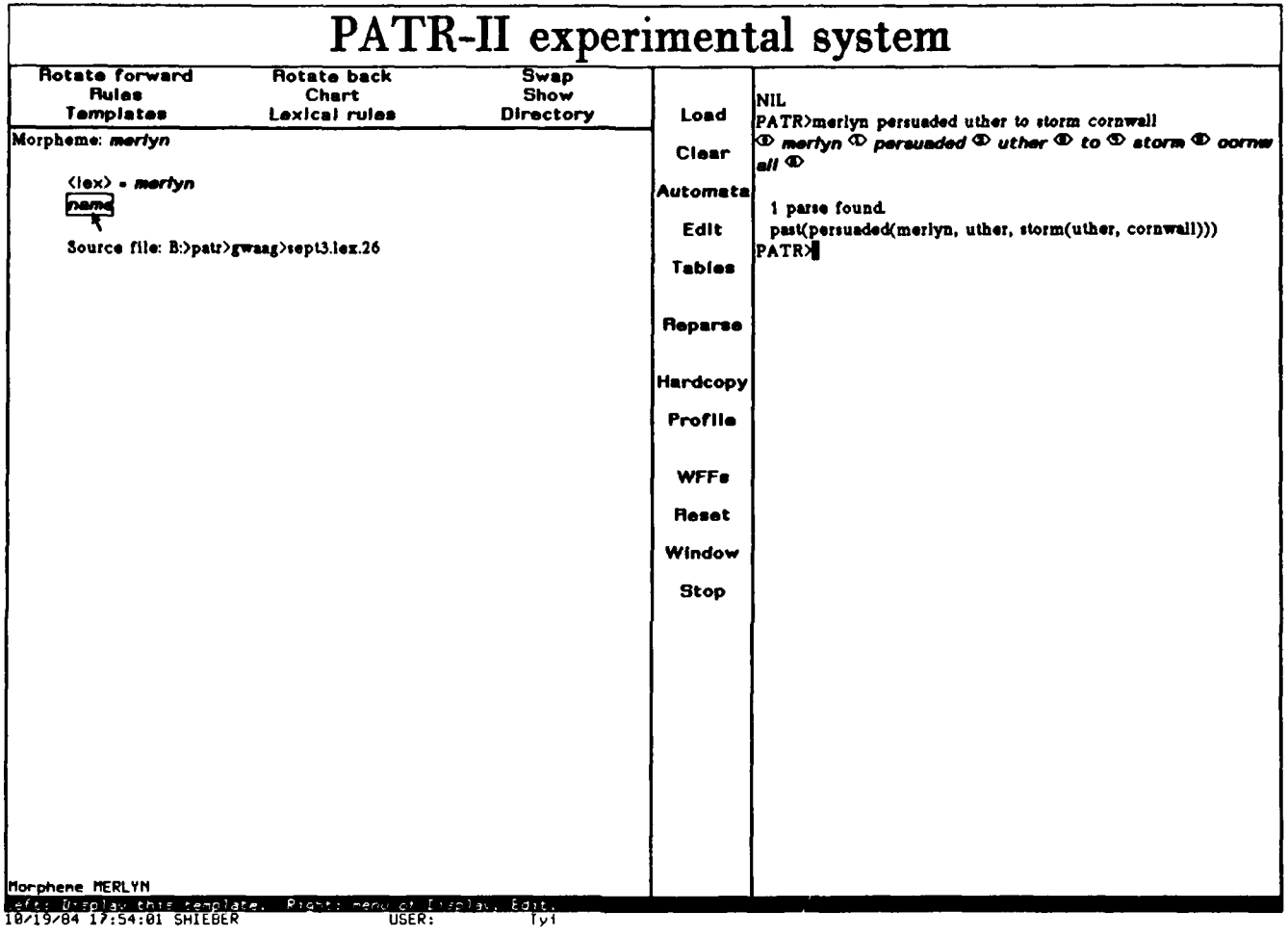
35. The chart is displayed in a display window. The user clicks on the word icon "merlyn" to display it.



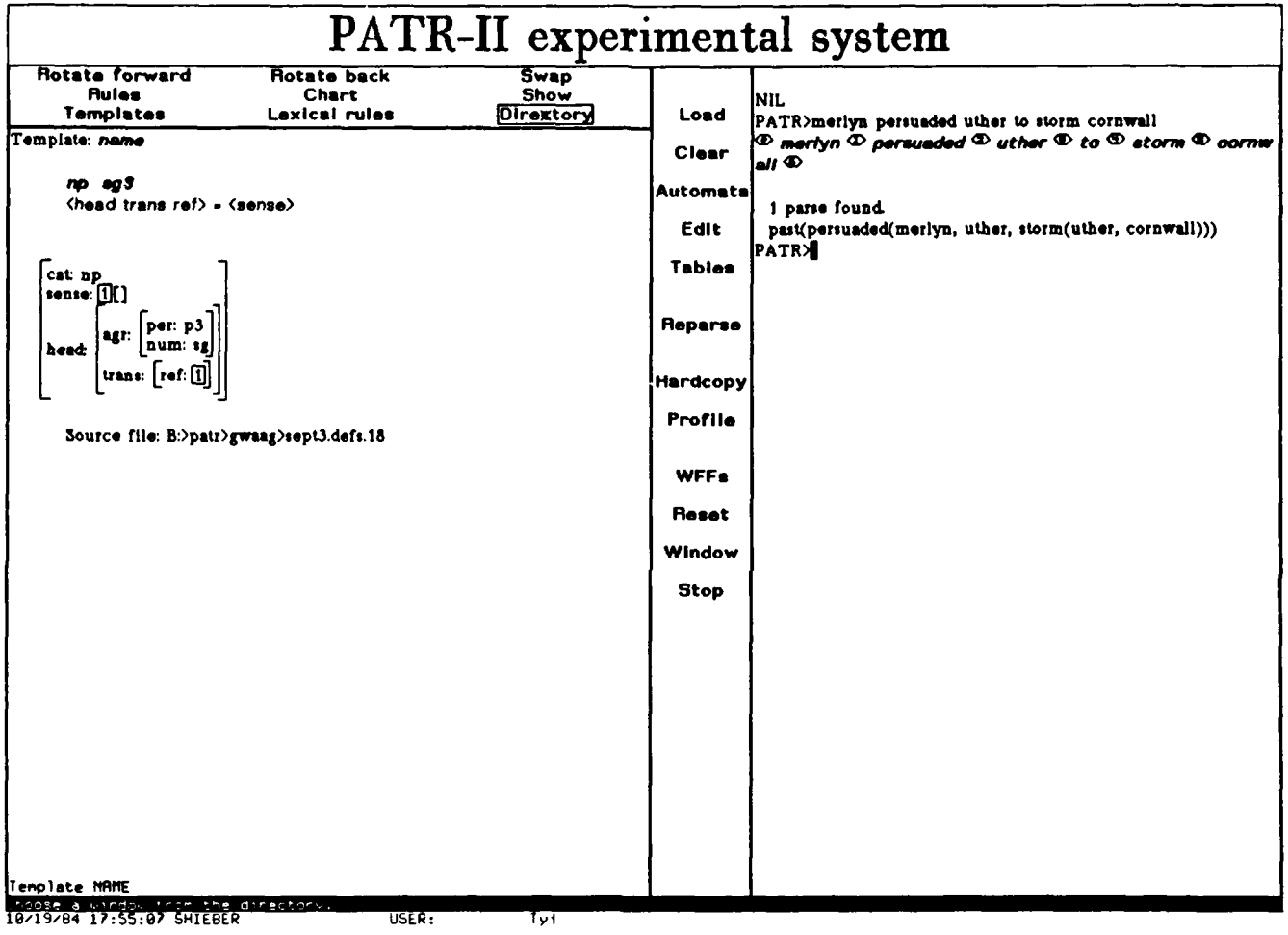
36. The word has only one sense; it is chosen.



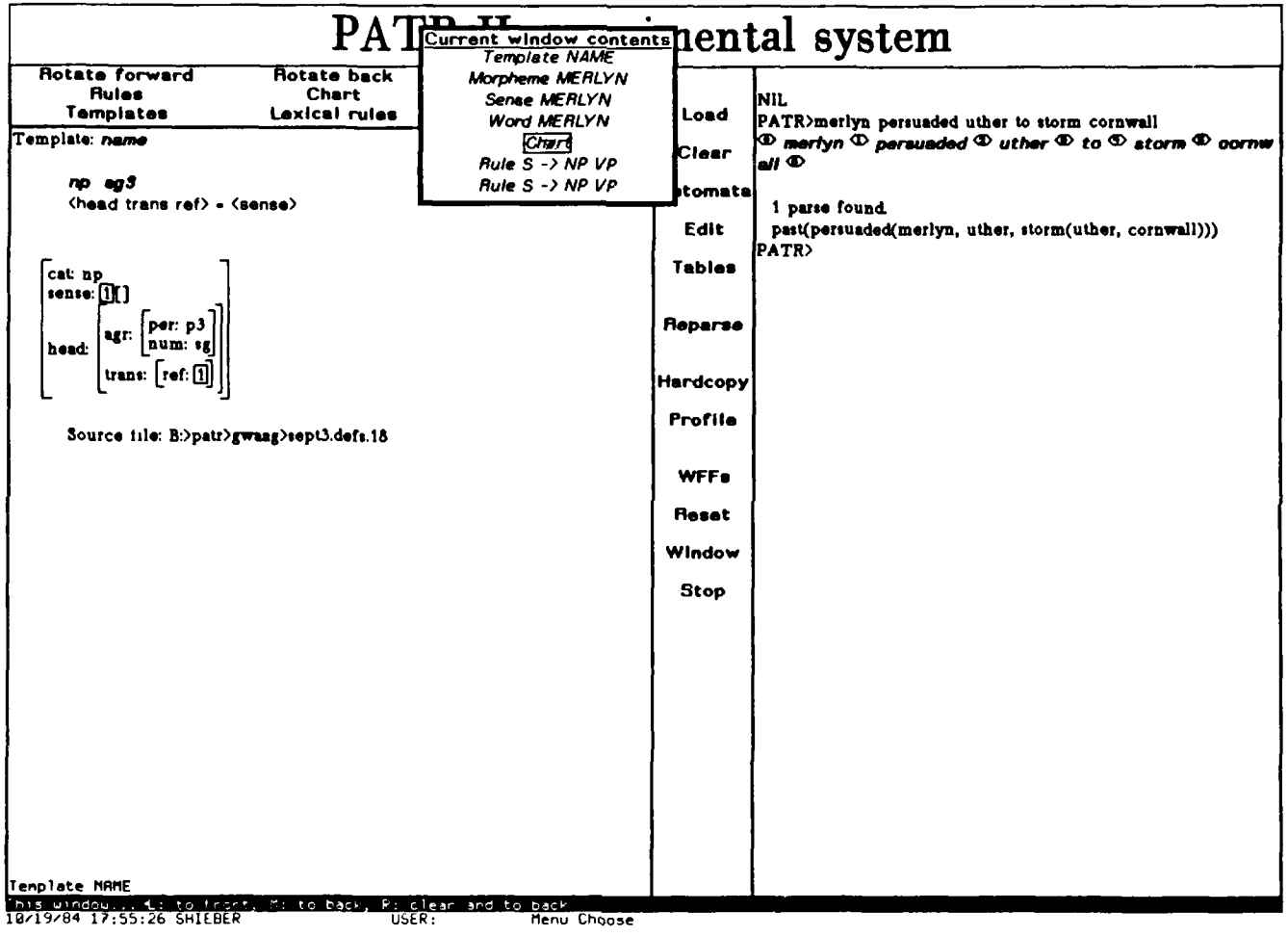
37. The information for that sense of the word is displayed. It has been derived by the morphological analyzer from two morphemes. The first one is clicked on.



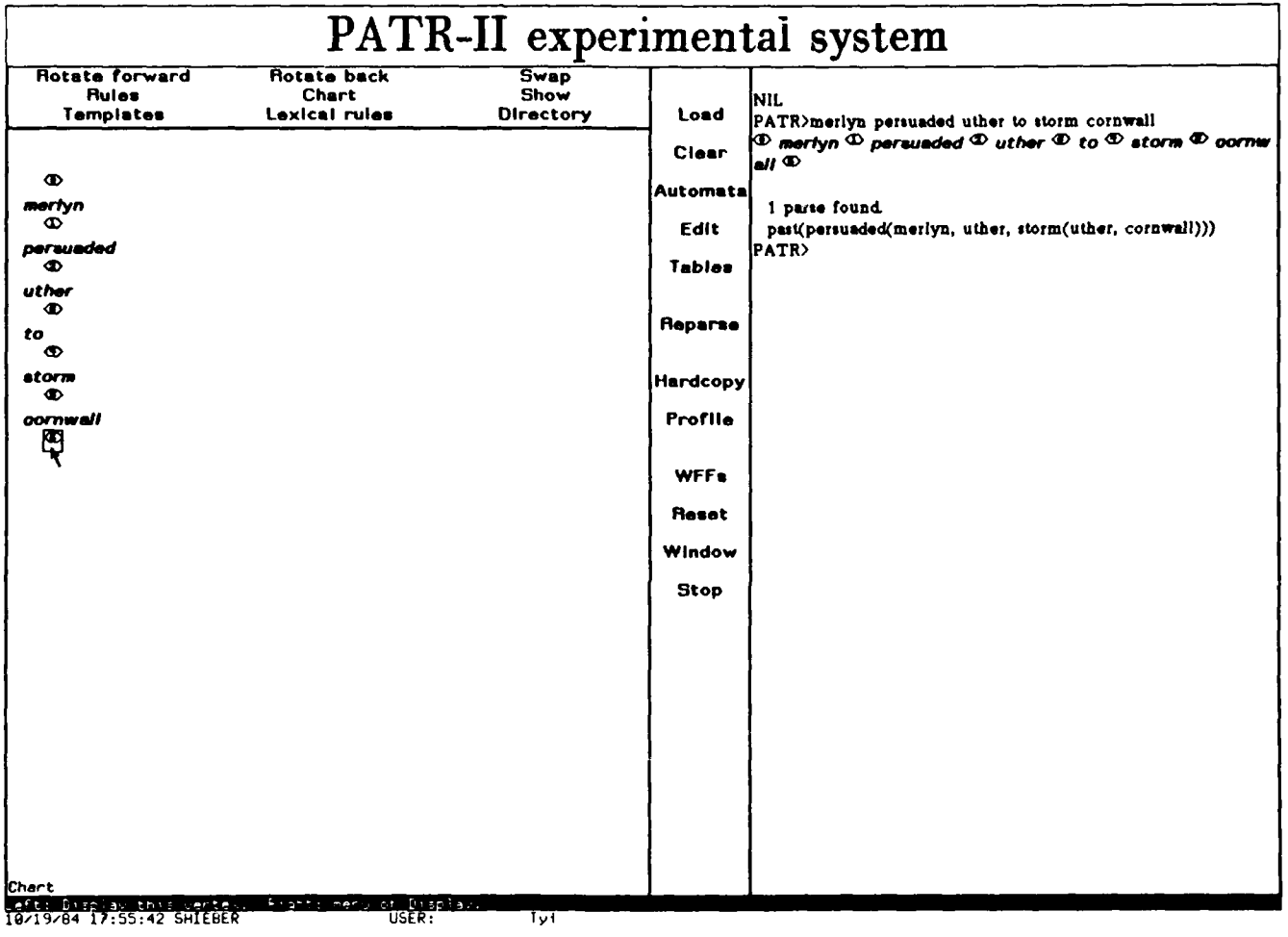
38. The information for this morpheme is displayed. This information comes directly from the lexical entry in the "sept3.lex" file. The definition makes use of the template "Name". The user clicks on an icon for that template.



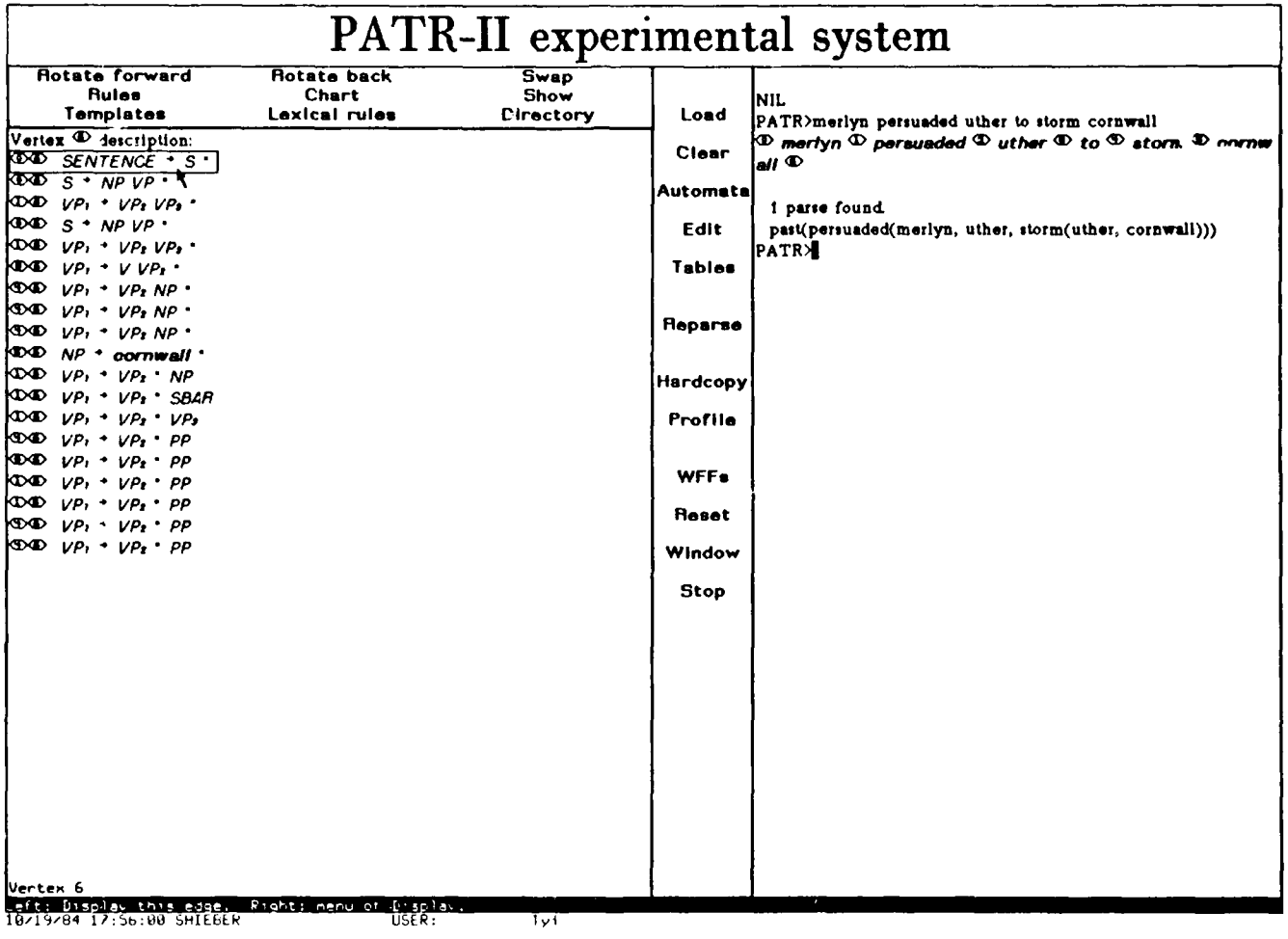
39. The template is displayed. The "Directory" command is about to be clicked on.



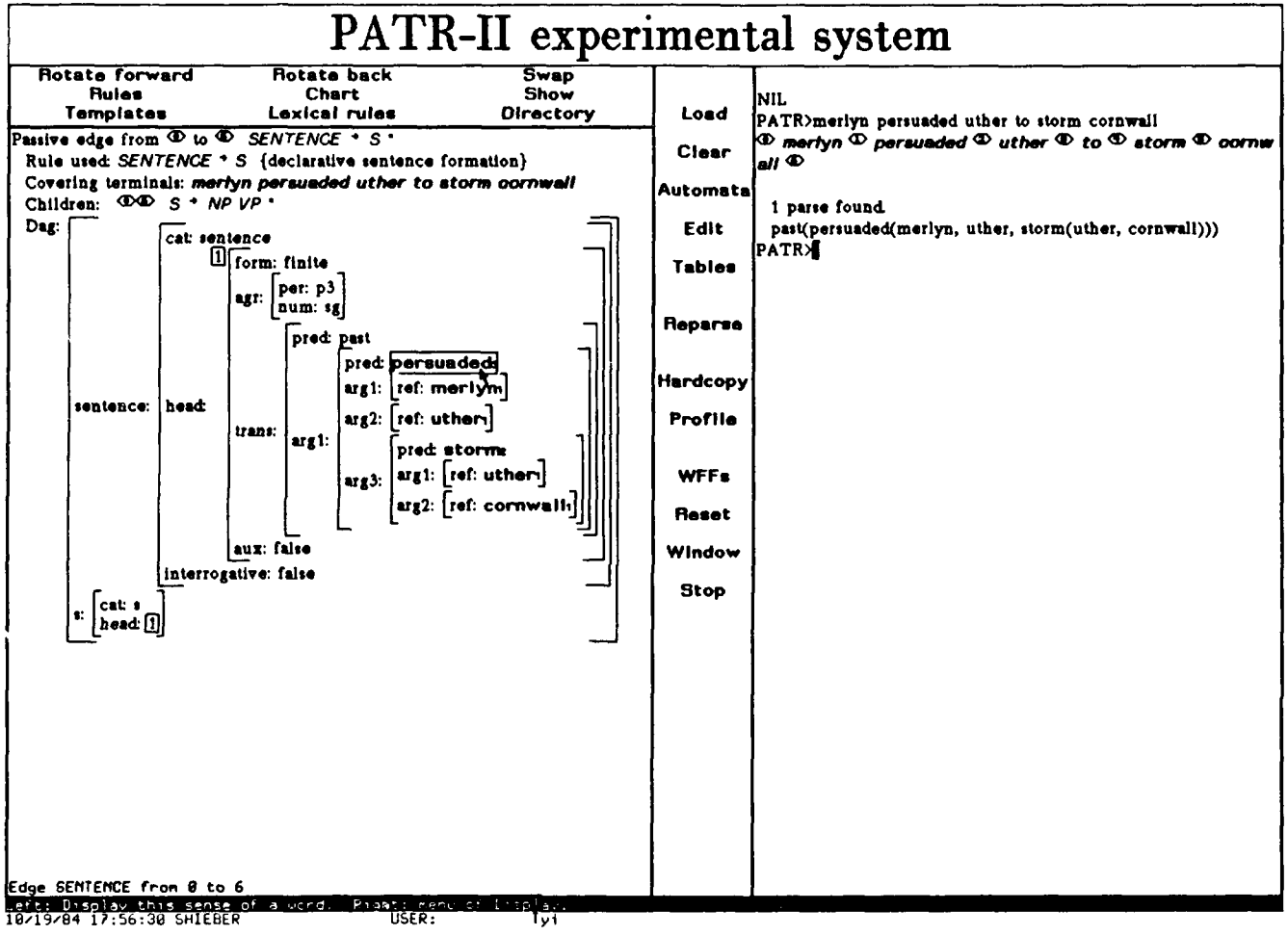
40. The directory shows the seven most recent windows of information, in order of recency. The user pulls up the chart window to the front by clicking left on it.



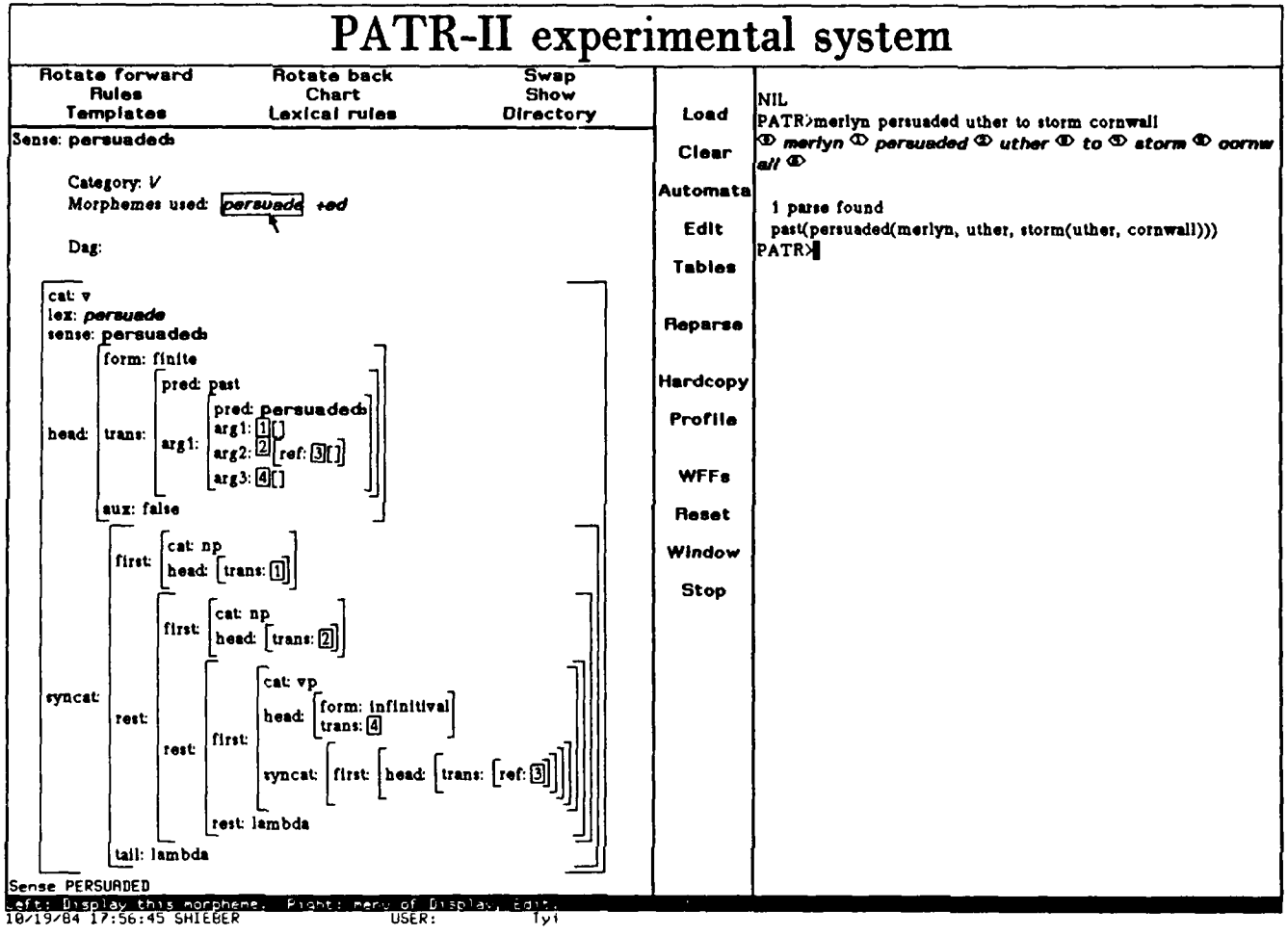
41. The chart window is moved to the front. Vertex 6 is about to be displayed.



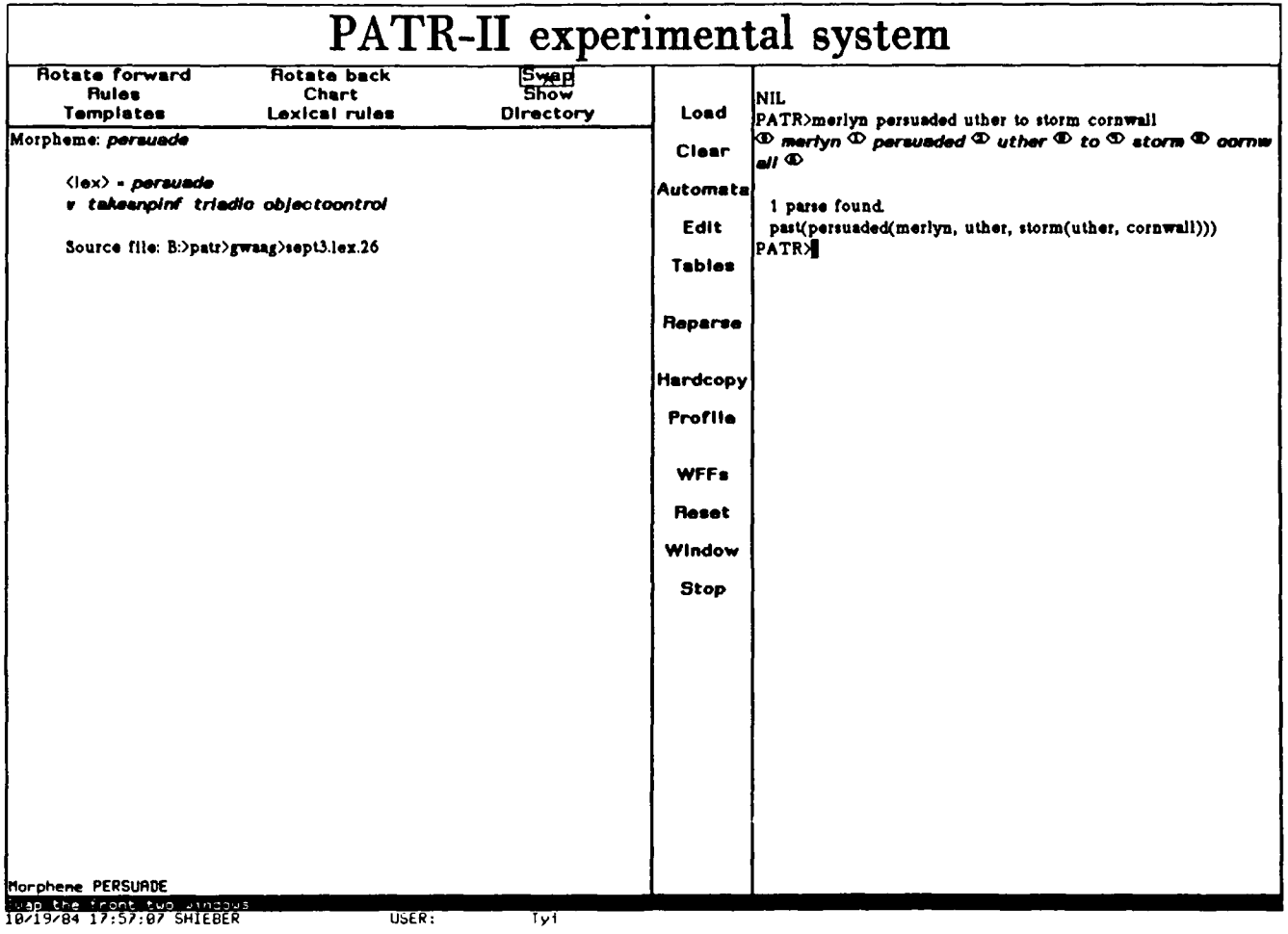
42. The vertex information is displayed. The user chooses an edge.



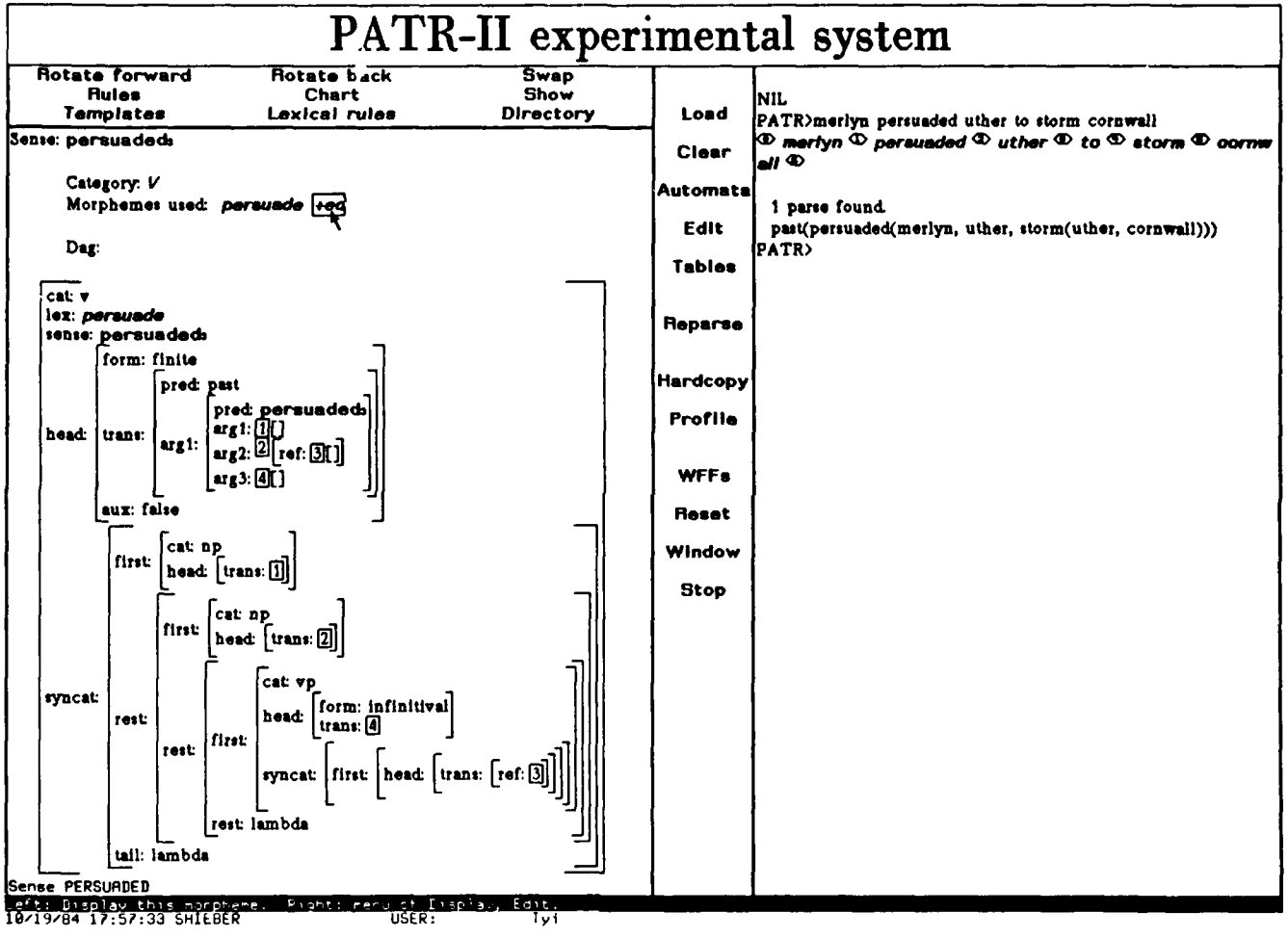
43. The edge is displayed. The user clicks on a sense icon incorporated in the DAG associated with the edge.



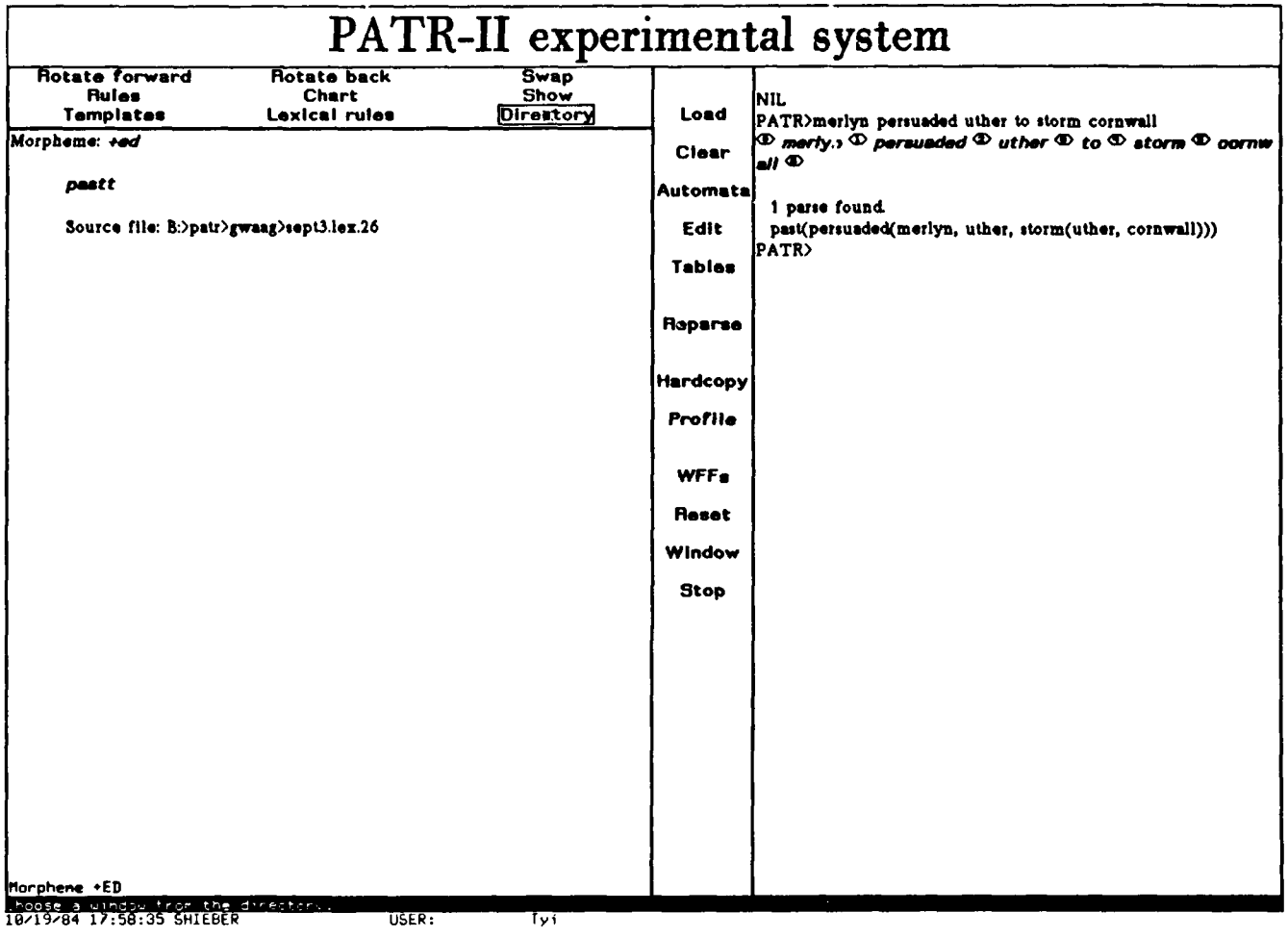
44. The sense of "persuaded" used in the final parse is displayed. The user looks at the first morpheme involved in building up the sense.



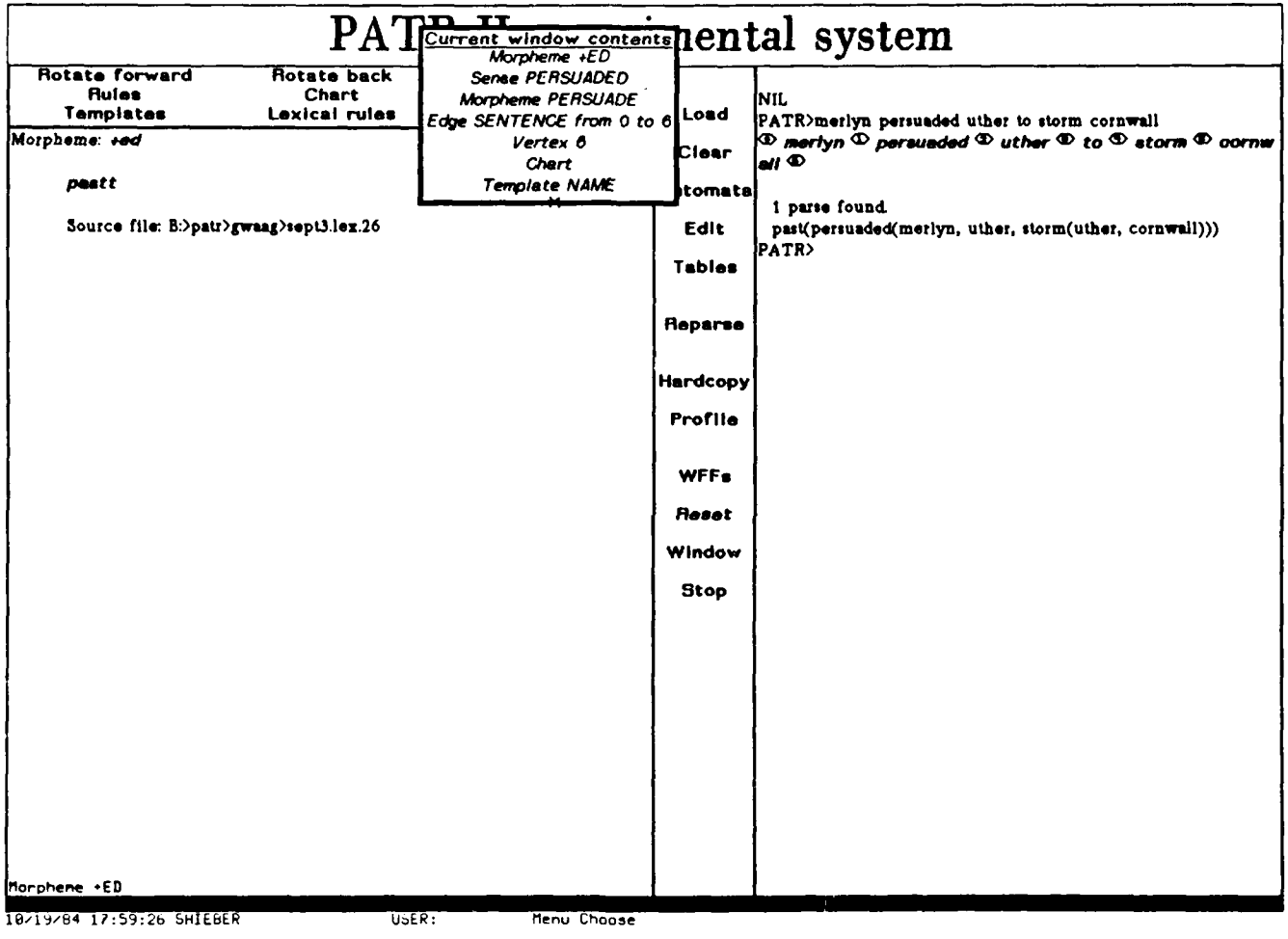
45. The morpheme is displayed. The user swaps the front two windows.



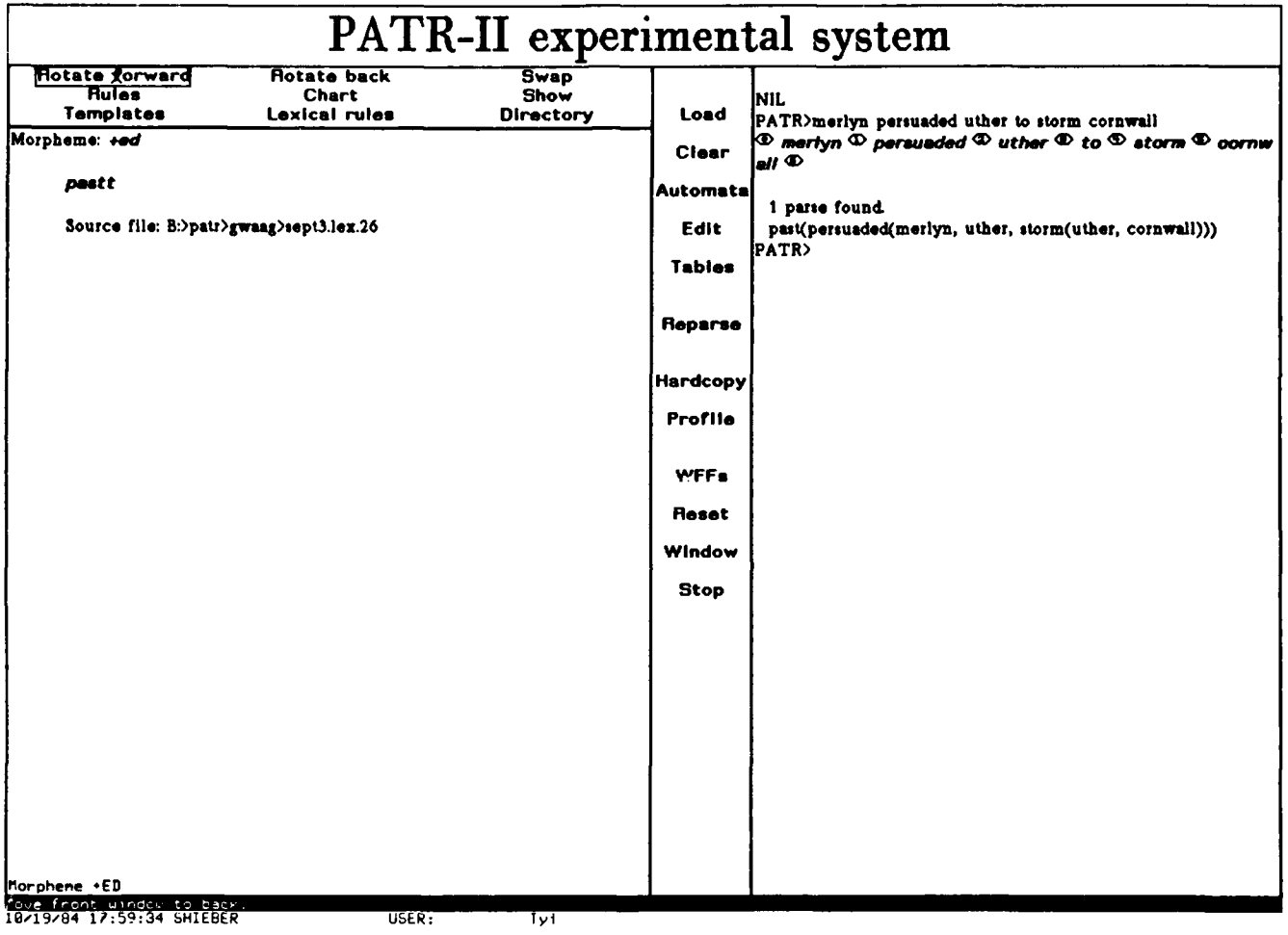
46. Back at the sense window, the user clicks on the other morpheme, the "ed" ending.



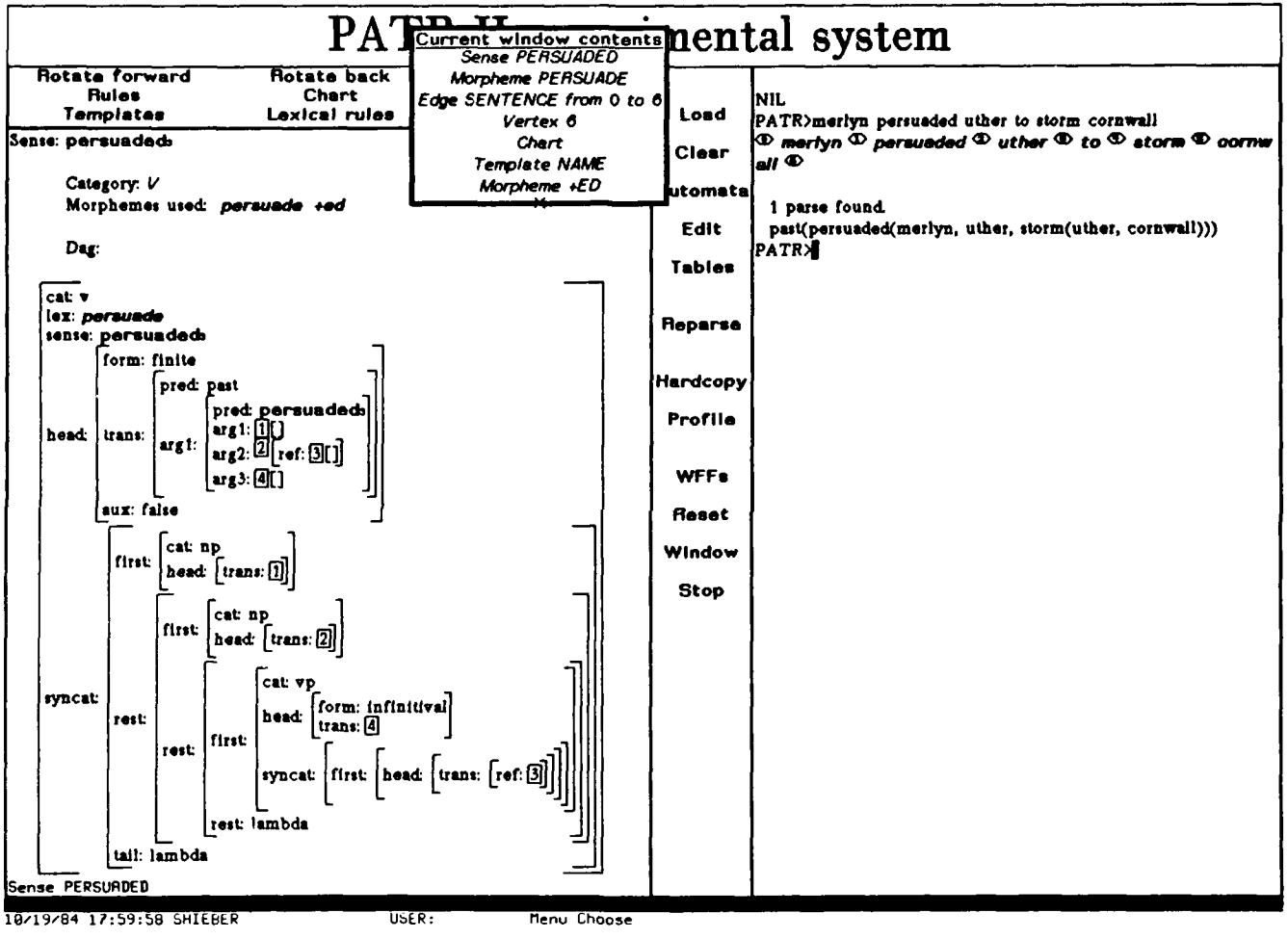
47. The ending is displayed.



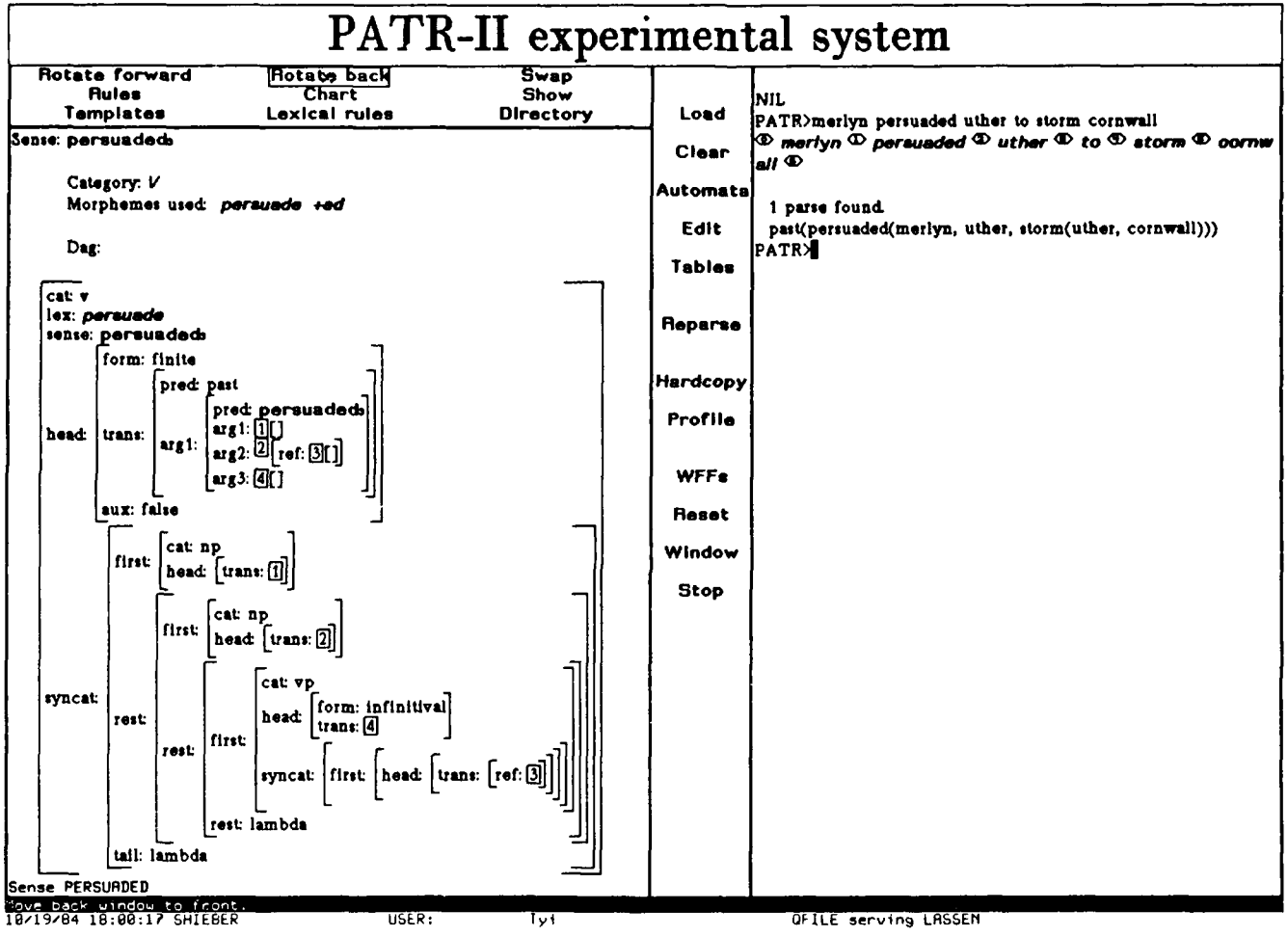
48. The directory shows the appropriate window contents.



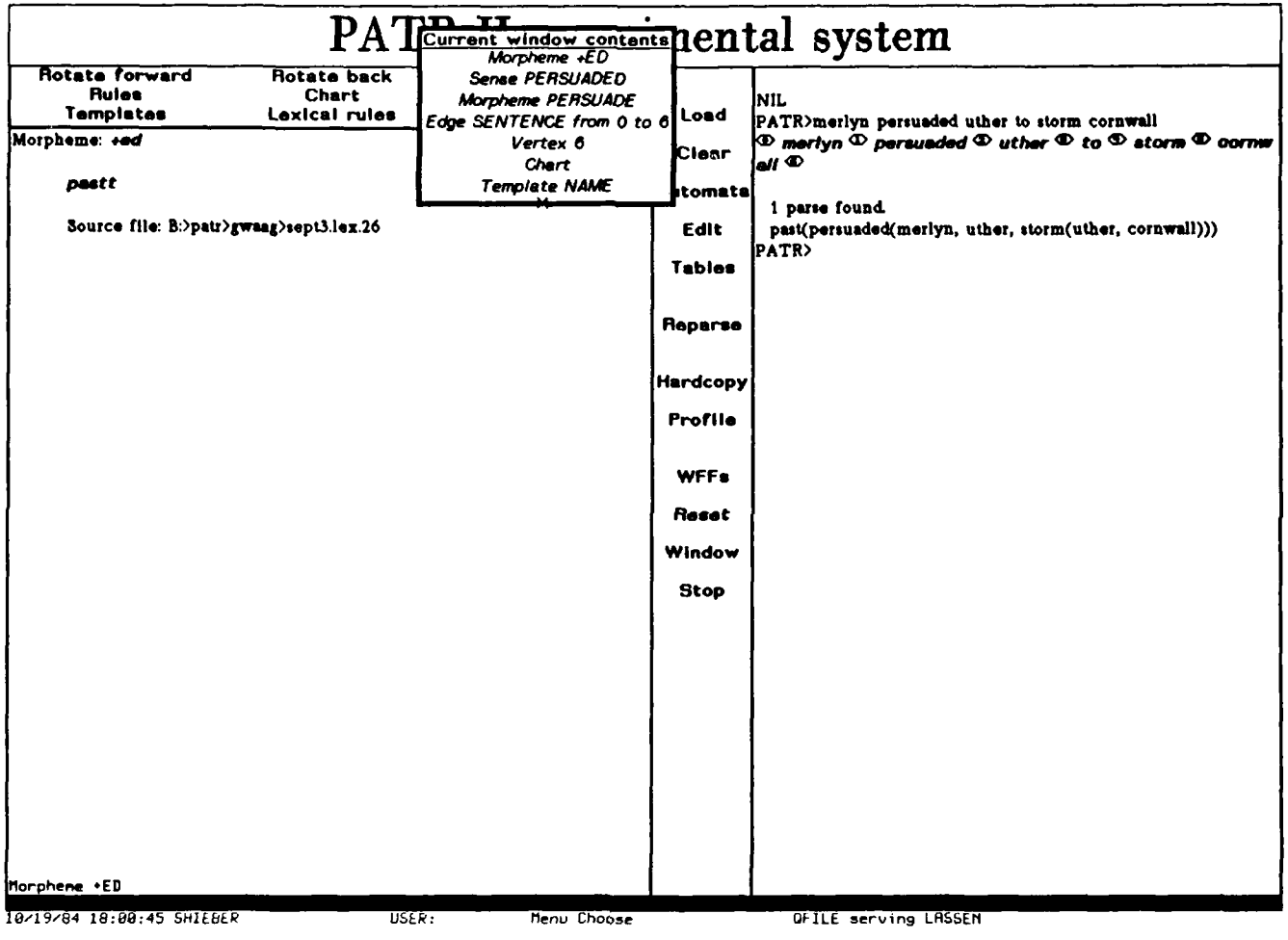
49. The user is about to click on the "Rotate forward" command.



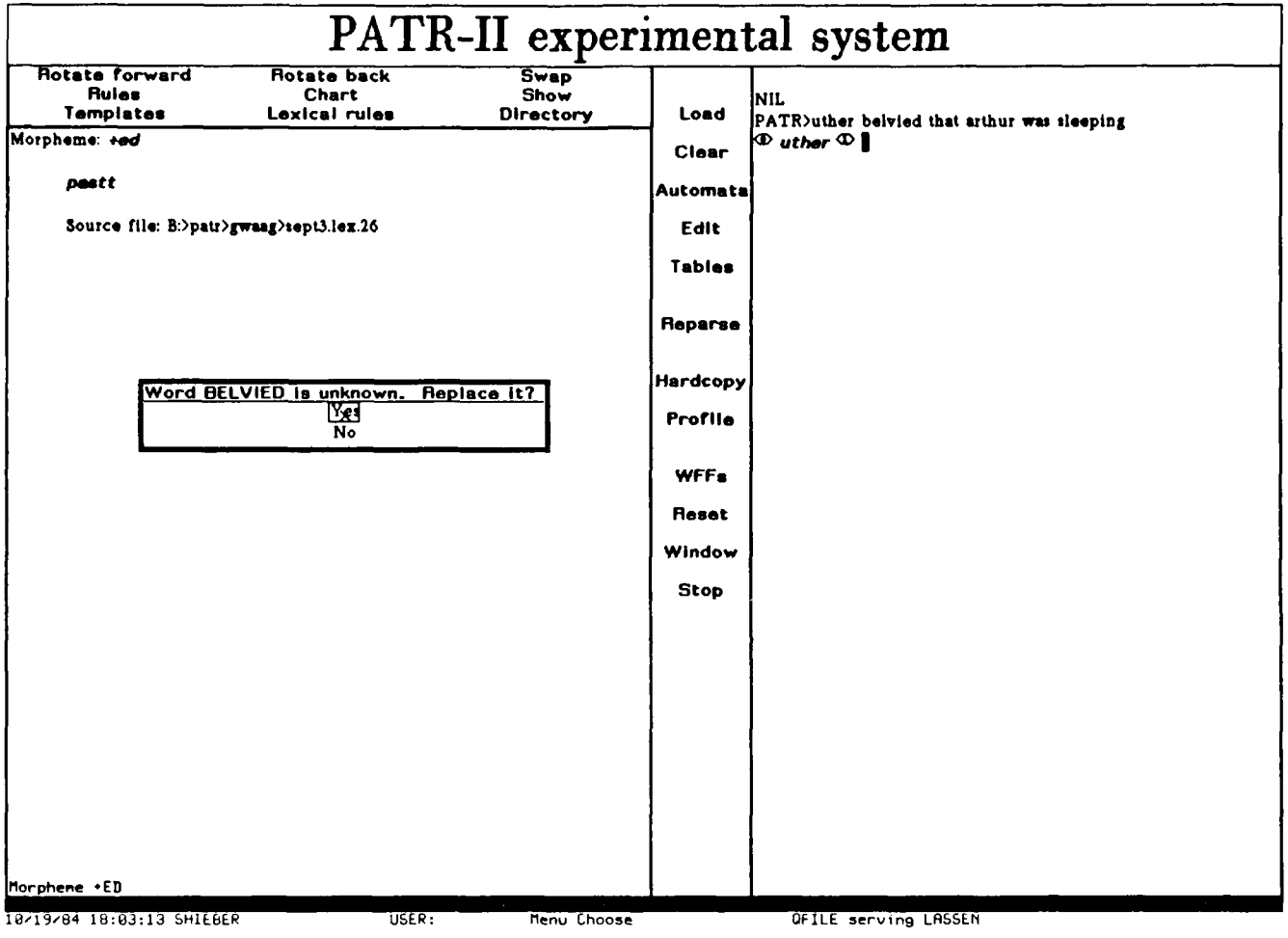
50. The directory shows that the windows have been rotated as expected. Notice that the sense window is now on top and displayed.



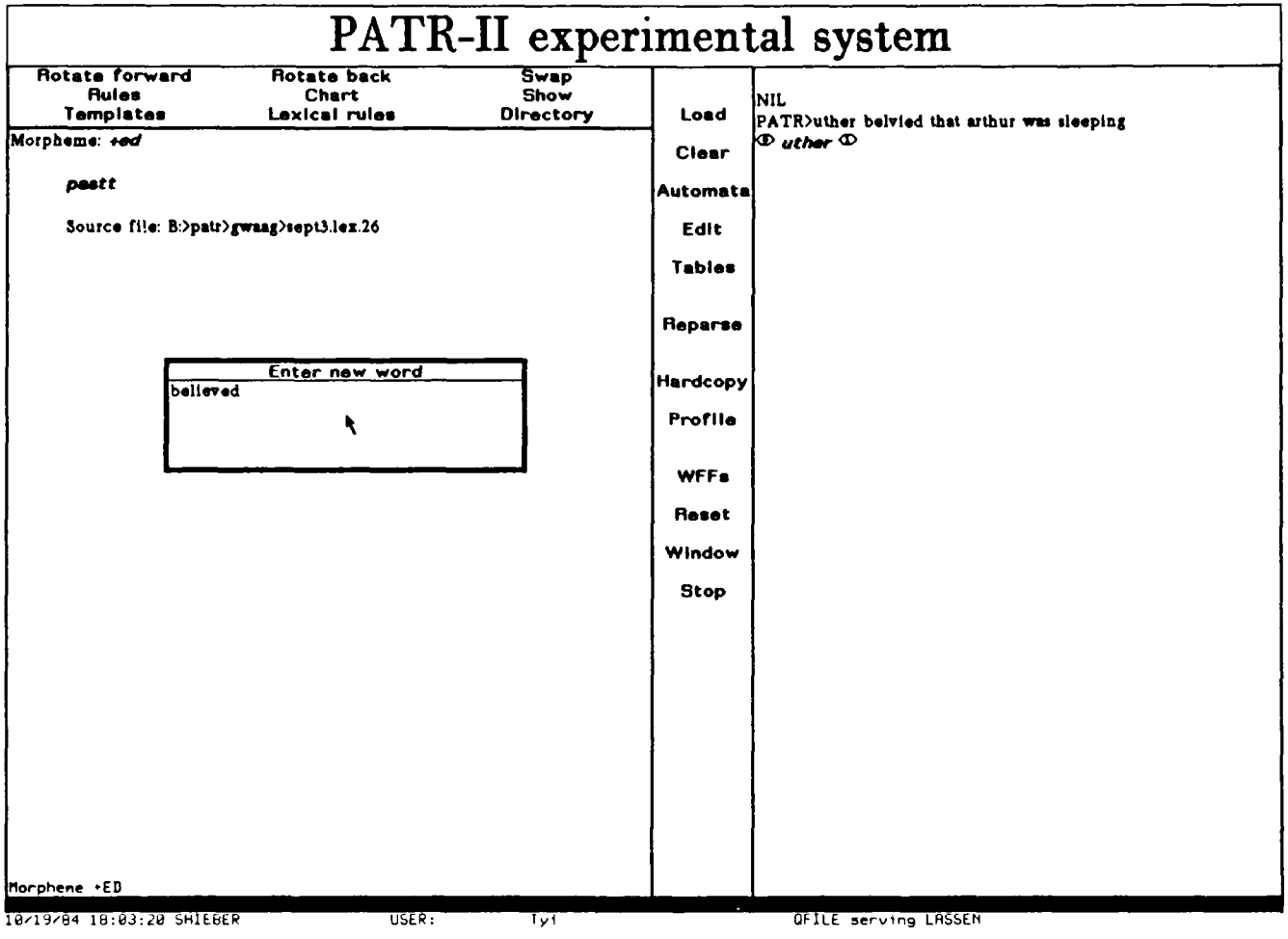
51. "Rotate back" works similarly.



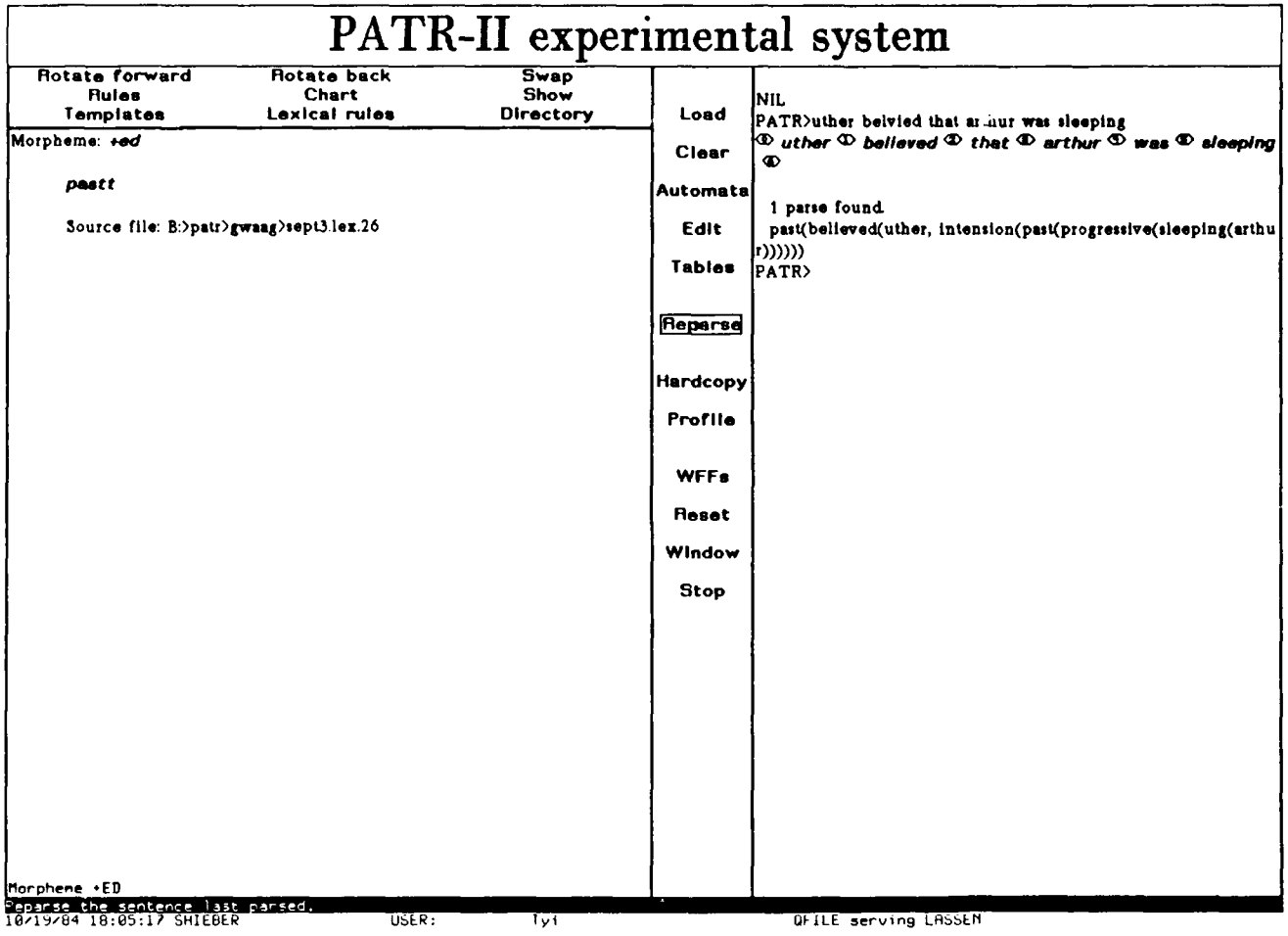
52. The windows are moved back to their original positions.



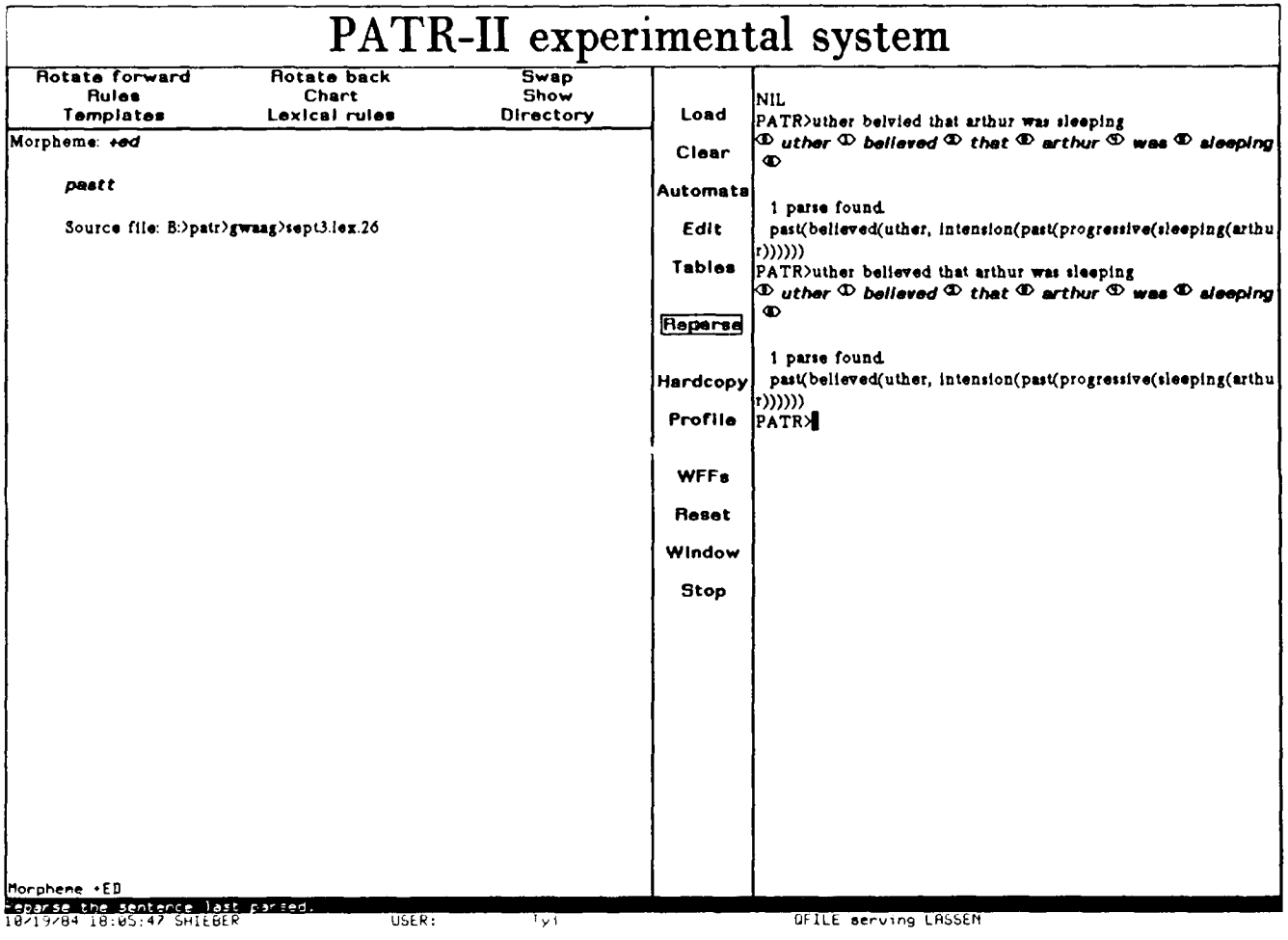
53. The left window is refreshed (by hitting the refresh key) and a new sentence is typed in. But the user has misspelled a word. The system asks if he wants to replace the word. The user is about to click on "Yes".



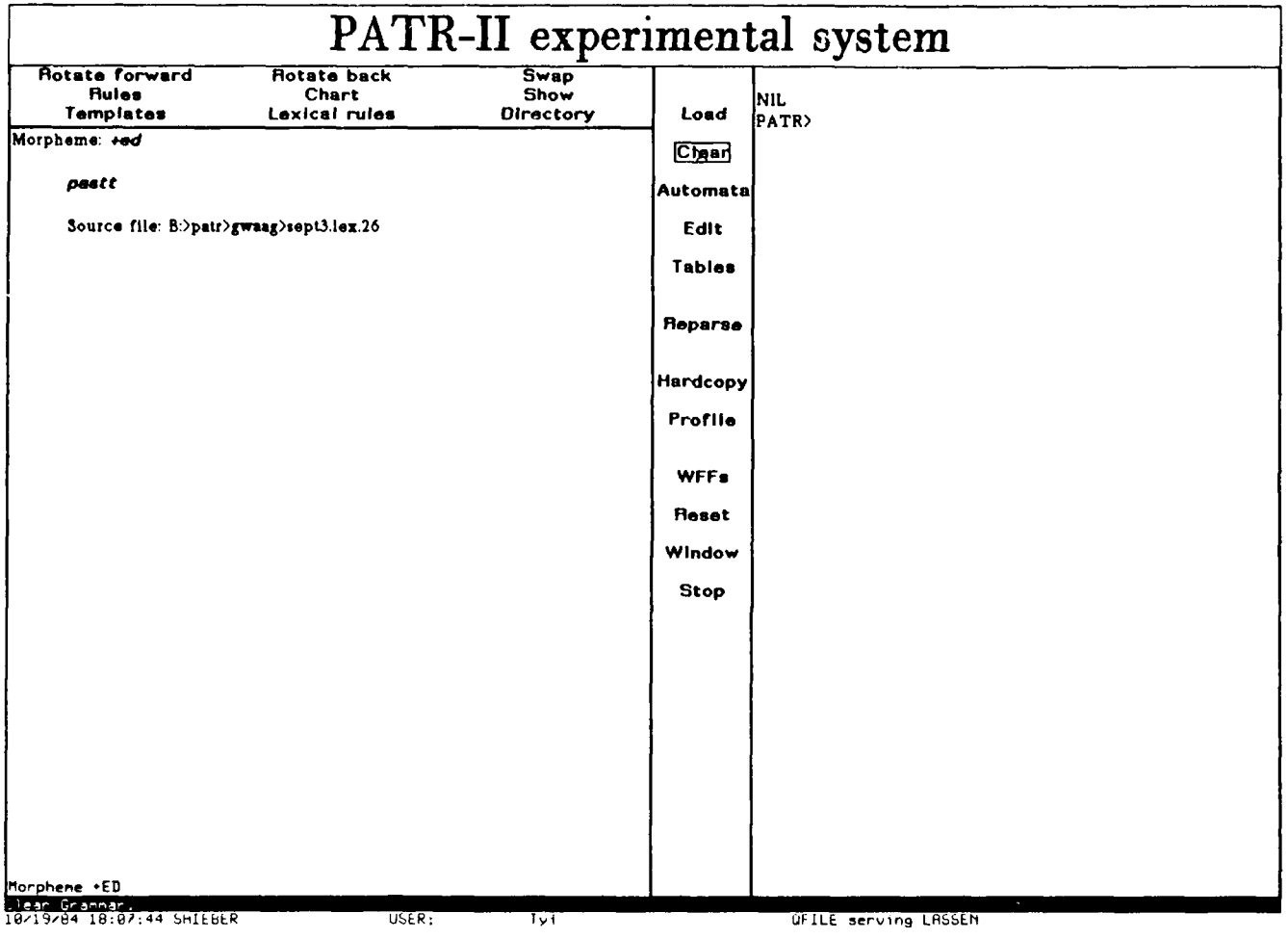
54. The user enters the new word to replace the misspelling.



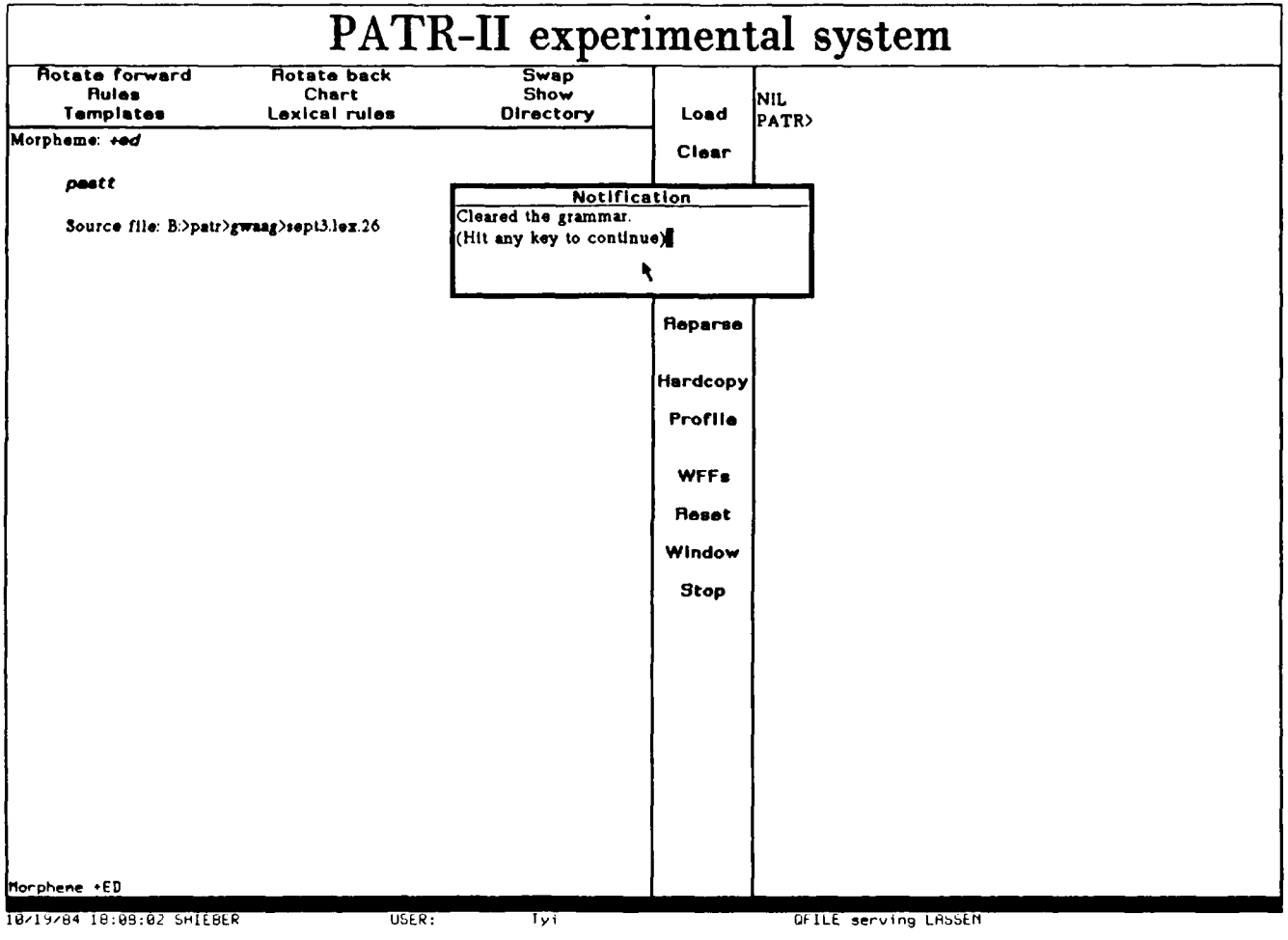
55. The parse continues, and a single parse is found. The user wants to reparse the sentence, so clicks on "Reparse".



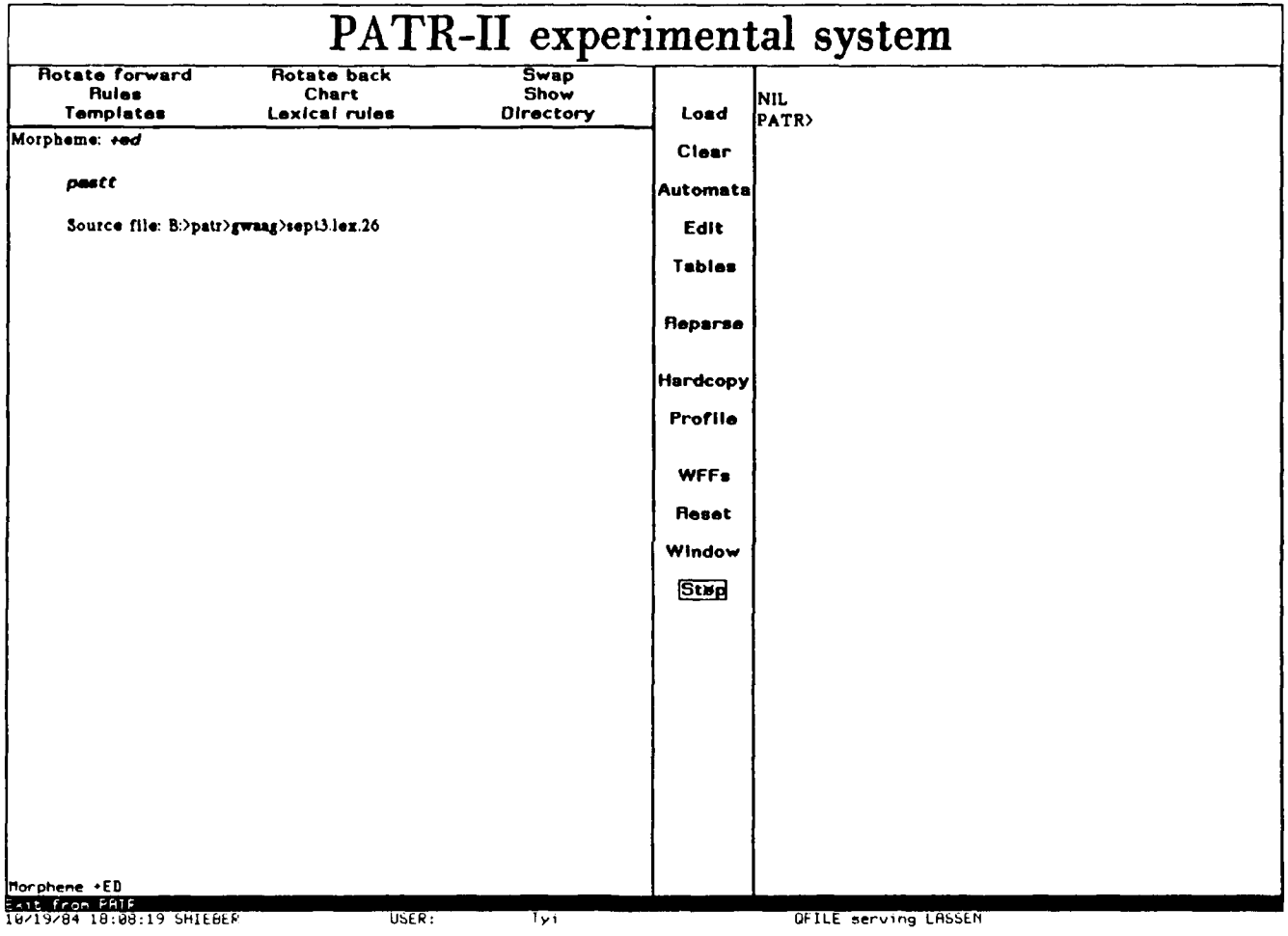
56. The sentence (with miss) elling corrected) is reparsed.



57. The screen is refreshed, and the user is about to clear the grammar.



58. The grammar information is cleared and a notification is put in a temporary window.



59. The user concludes the session by clicking on "Stop".

Chapter 3

A Structure-Sharing Representation for Unification-Based Grammar Formalisms

This chapter was written by Fernando Pereira.

3.1 Overview

In this chapter I describe a method, *structure sharing*, for the representation of complex phrase types in a parser for PATR-II, a unification-based grammar formalism.

In parsers for unification-based grammar formalisms, complex phrase types are derived by incremental refinement of the phrase types defined in grammar rules and lexical entries. In a naïve implementation, a new phrase type is built by copying older ones and then combining the copies according to the constraints stated in a grammar rule. The structure-sharing method eliminates most such copying by representing updates to objects (phrase types) separately from the objects themselves.

The present work is inspired by the structure-sharing method for theorem proving introduced by Boyer and Moore [1] and on the variant of it that is used in some Prolog implementations [9].

3.2 Grammars with Unification

The data representation discussed here is applicable, with but minor changes, to a variety of grammar formalisms based on unification, such as definite-clause grammars [6], functional-unification grammar [4], lexical-functional grammar [4] and PATR-II [12]. For the sake of

concreteness, however, our discussion will be in terms of the PATR-II formalism.

The basic idea of unification-based grammar formalisms is very simple. As with context-free grammars, grammar rules state how phrase types combine to yield other phrase types. But whereas a context-free grammar allows only a finite number of predefined atomic phrase types or *nonterminals*, a unification-based grammar will in general define implicitly an infinity of phrase types.

A phrase type is defined by a set of constraints. A grammar rule is a set of constraints between the type X_0 of a phrase and the types $X_1, \ldots, X_n$ of its constituents. The rule may be applied to the analysis of a string s_0 as the concatenation of constituents $s_1, \ldots, s_n$ if and only if the types of the s_i are compatible with the types X_i and the constraints in the rule.

Unification is the operation that determines whether two types are compatible by building the most general type compatible with both.

If the constraints are equations between attributes of phrase types, as is the case in PATR-II, two phrase types can be unified whenever they do not assign distinct values to the same attribute. The unification is then just the conjunction (set union) of the corresponding sets of constraints [10].

Here is a sample rule, in a simplified version of the PATR-II notation:

$$\begin{array}{lll} X_0 \rightarrow X_1\ X_2: & \langle X_0\ cat \rangle & = S \\ & \langle X_1\ cat \rangle & = NP \\ & \langle X_2\ cat \rangle & = VP \\ & \langle X_1\ agr \rangle & = \langle X_2\ agr \rangle \\ & \langle X_0\ trans \rangle & = \langle X_2\ trans \rangle \\ & \langle X_0\ trans\ arg_1 \rangle & = \langle X_1\ trans \rangle \end{array} \tag{3.1}$$

This rule may be read as stating that a phrase of type X_0 can be the concatenation of a phrase of type X_1 and a phrase of type X_2, provided that the attribute equations of the rule are satisfied if the phrases are substituted for their types. The equations state that phrases of types X_0, X_1, and X_2 have categories S, NP, and VP, respectively, that types X_1 and X_2 have the same agreement value, that types X_0 and X_2 have the same translation, and that the first argument of X_0's translation is the translation of X_1.

Formally, the expressions of the form $\langle l_1 \cdots l_m \rangle$ used in attribute equations are *paths* and each l_i is a *label*.

When all the phrase types in a rule are given constant *cat* (category) values by the rule, we can use an abbreviated notation in which the phrase type variables X_i are replaced by their category values and the category-setting equations are omitted. For example, rule (3.1) may be written as

$$\begin{array}{lll} S \rightarrow NP\ VP: & \langle NP\ agr \rangle & = \langle VP\ agr \rangle \\ & \langle S\ trans \rangle & = \langle VP\ trans \rangle \\ & \langle S\ trans\ arg_1 \rangle & = \langle NP\ trans \rangle \end{array} \tag{3.2}$$

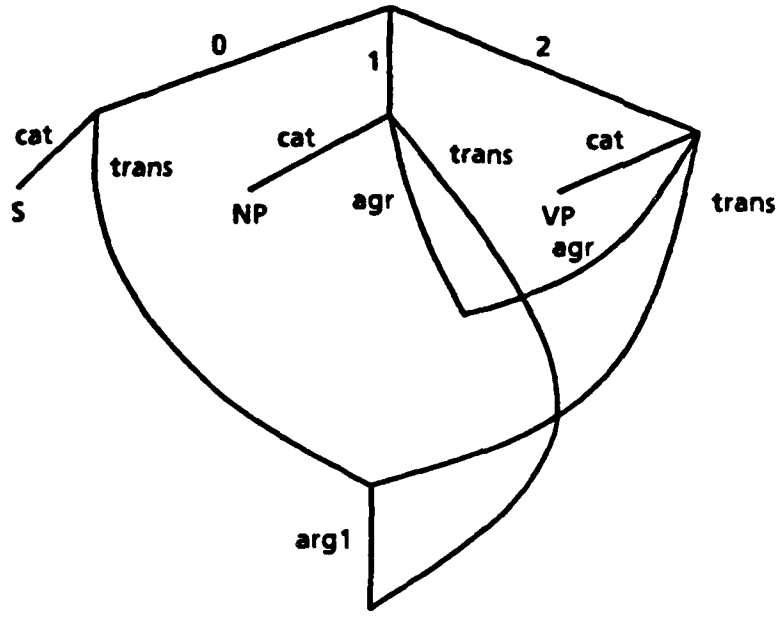


Figure 3.1: *DAG Representation of a Rule*

In existing PATR-II implementations, phrase types are not actually represented by their sets of defining equations. Instead, they are represented by symbolic solutions of the equations in the form of directed acyclic graphs (*DAGs*) with arcs labeled by the attributes used in the equations. DAG nodes represent the values of attributes and an arc labeled by l goes from node m to node n if and only if, according to the equations, the value represented by m has n as the value of its l attribute [10].

A DAG node (and by extension a DAG) is said to be *atomic* if it represents a constant value; *complex* if it has some outgoing arcs; and a *leaf* if is is neither atomic or complex, that is, if it represents an as yet completely undetermined value. The *domain* dom(d) of a complex DAG d is the set of labels on arcs leaving the top node of d. Given a DAG d and a label $l \in \text{dom}(d)$ we denote by d/l the sub-DAG of d at the end of the arc labeled l from the top node of d. By extension, for any path p whose labels are in the domains of the appropriate sub-DAGs, d/p represents the sub-DAG of d at the end of path p from the root of d.

For uniformity, lexical entries and grammar rules are also represented by appropriate DAGs. For example, the DAG for rule (3.1) is shown in Figure 3.1.

3.3 The Problem

In a chart parser [3] all the intermediate stages of derivations are encoded in *edges*, representing either incomplete (*active*) or complete (*passive*) phrases. For PATR-II, each edge

contains a DAG instance that represents the phrase type of that edge. The problem we address here is how to encode multiple DAG instances efficiently.

In a chart parser for context-free grammars, the solution is trivial: instances can be represented by the unique internal names (that is, addresses) of their objects because the information contained in an instance is exactly the same as that in the original object.

In a parser for PATR-II or any other unification-based formalism, however, distinct instances of an object will in general specify different values for attributes left *unspecified* in the original object. Clearly, the attribute values specified for one instance are independent of those for another instance of the same object.

One obvious solution is to build new instances by copying the original object and then updating the copy with the new attribute values. This was the solution adopted in the first PATR-II parser [12]. The high cost of this solution both in time spent copying and in space required for the copies themselves constitutes the principal justification for employing the method described here.

3.4 Structure Sharing

Structure sharing is based on the observation that an initial object, together with a list of update records, contains the same information as the object that results from applying the updates to the initial object. In this way, we can trade the cost of actually applying the updates (with possible copying to avoid the destruction of the source object) against the cost of having to compute the effects of updates when examining the derived object. This reasoning applies in particular to DAG instances that are the result of adding attribute values to other instances.

As in the variant of Boyer and Moore's method [1] used in Prolog [9], I shall represent a DAG instance by a *molecule* (see Figure 3.2) consisting of

1. [A pointer to] the initial DAG, the instance's *skeleton*
2. [A pointer to] a table of updates of the skeleton, the instance's *environment*.

Environments may contain two kinds of updates: *reroutings* that replace a DAG node with another DAG; *arc bindings* that add to a node a new outgoing arc pointing to a DAG. Figure 3.3 shows the unification of the DAGs

$$\begin{aligned} I_1 &= [a : x, b : y] \\ I_2 &= [c : [d : e]] \end{aligned}$$

After unification, the top node of I_2 is rerouted to I_1 and the top node of I_1 gets an arc binding with label c and a value that is the sub-DAG $[d : e]$ of I_2. As we shall see later, any update of a DAG represented by a molecule is either an update of the molecule's skeleton or an update of a DAG (to which the same reasoning applies) appearing in the molecule's

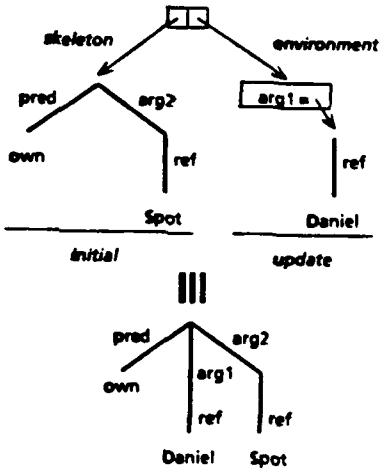


Figure 3.2: Molecule

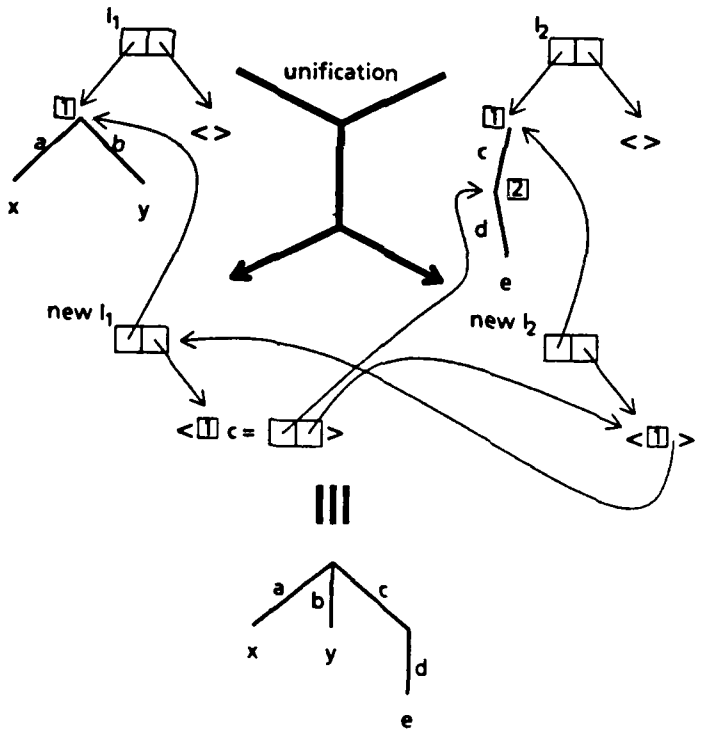


Figure 3.3: Unification of Two Molecules

enviroment. Therefore, the updates in a molecule's environment are always shown in figures tagged by a boxed number identifying the affected node in the molecule's skeleton.

The choice of which DAG is rerouted and which one gets arc bindings is arbitrary.

For reasons discussed later, the cost of looking up instance node updates in Boyer and Moore's environment representation is $O(|d|)$, where $|d|$ is the length of the derivation (a sequence of resolutions) of the instance. In the present representation, however, this cost is only $O(\log |d|)$. This better performance is achieved by particularizing the environment representation and by splitting the representational scheme into two components: a *memory organization* and a *DAG representation.*

A DAG representation is a way of mapping the *mathematical entity* DAG onto a memory. A memory organization is a way of putting together a memory that has certain properties with respect to lookup, updating and copying. One can think of the memory organization as the hardware and the DAG representation as the data structure.

3.5 Memory Organization

In practice, random-access memory can be accessed and updated in constant time. However, updates destroy old values, which is obviously unacceptable when dealing with alternative updates of the same data structure. If we want to keep the old version, we need to copy it first into a separate part of memory and change the copy instead. For the normal kind of memory, copying time is proportional to the size of the object copied.

The present scheme uses another type of memory organization — *virtual-copy arrays* — which requires $O(\log n)$ time to access or update an array with highest used index of n, but in which the old contents are not destroyed by updating. Virtual-copy arrays were developed by David H. D. Warren [10] as an implementation of extensible arrays for Prolog.

Virtual-copy arrays provide a fully general memory structure: anything that can be stored in random-access memory can be stored in virtual-copy arrays, although pointers in machine memory correspond to indexes in a virtual-copy array. An updating operation takes a virtual-copy array, an index, and a new value and returns a new virtual-copy array with the new value stored at the given index. An access operation takes an array and an index, and returns the value at that index.

Basically, virtual-copy arrays are 2^k-ary trees for some fixed $k > 0$. Define the *depth* $d(n)$ of a tree node n to be 0 for the root and $d(p) + 1$ if p is the parent of n. Each virtual-copy array a has also a positive *depth* $D(a) \geq \max\{d(n) : n \text{ is a node of } a\}$. A tree node at depth $D(a)$ (necessarily a leaf) can be either an array element or the special marker $\perp$ for unassigned elements. All leaf nodes at depths lower than $D(a)$ are also $\perp$, indicating that no elements have yet been stored in the subarray below the node. With this arrangement, the array can store at most $2^{kD(a)}$ elements, numbered 0 through $2^{kD(a)} - 1$, but unused subarrays need not be allocated.

By numbering the 2^k daughters of a nonleaf node from 0 to $2^k - 1$, a path from a's root to an array element (a leaf at depth $D(a)$) can be represented by a sequence $n_0 \cdots n_{D(a)-1}$

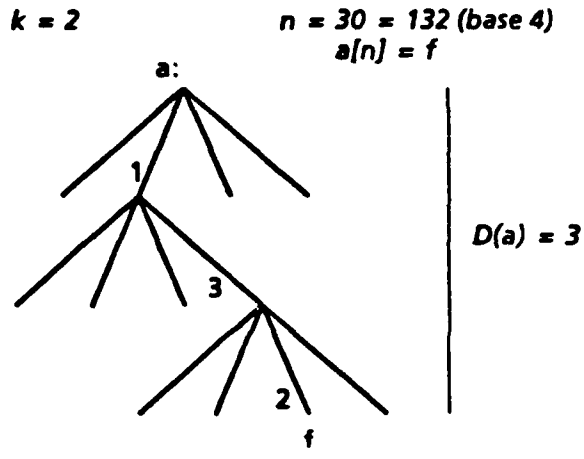


Figure 3.4: Virtual-Copy Array

in which n_d is the number of the branch taken at depth d. This sequence is just the base 2^k representation of the index n of the array element, with n_0 the most significant digit and $n_{D(a)}$ the least significant (Figure 3.4).

When a virtual-copy array a is updated, one of two things may happen. If the index for the updated element exceeds the maximum for the current depth (as in the $a[8] := g$ update in Figure 3.5), a new root node is created for the updated array and the old array becomes the leftmost daughter of the new root. Other nodes are also created, as appropriate, to reach the position of the new element. If, on the other hand, the index for the update is within the range for the current depth, the path from the root to the element being updated is copied and the old element is replaced in the new tree by the new element (as in the $a[2] := h$ update in Figure 3.5). This description assumes that the element being updated has already been set. If not, the branch to the element may terminate prematurely in a $\perp$ leaf, in which case new nodes are created to the required depth and attached to the appropriate position at the end of the new path from the root.

3.6 DAG Representation

Any DAG representation can be implemented with virtual-copy memory instead of random-access memory. If that were done for the original PATR-II copying implementation, a certain measure of structure sharing would be achieved.

The present scheme, however, goes well beyond that by using the method of structure sharing introduced in Section 3.4. As we saw there, an instance object is represented by a molecule, a pair consisting of a skeleton DAG (from a rule or lexical entry) and an update environment. We shall now examine the structure of environments.

In a chart parser for PATR-II, DAG instances in the chart fall into two classes.

Base instances are those associated with edges that are created directly from lexical

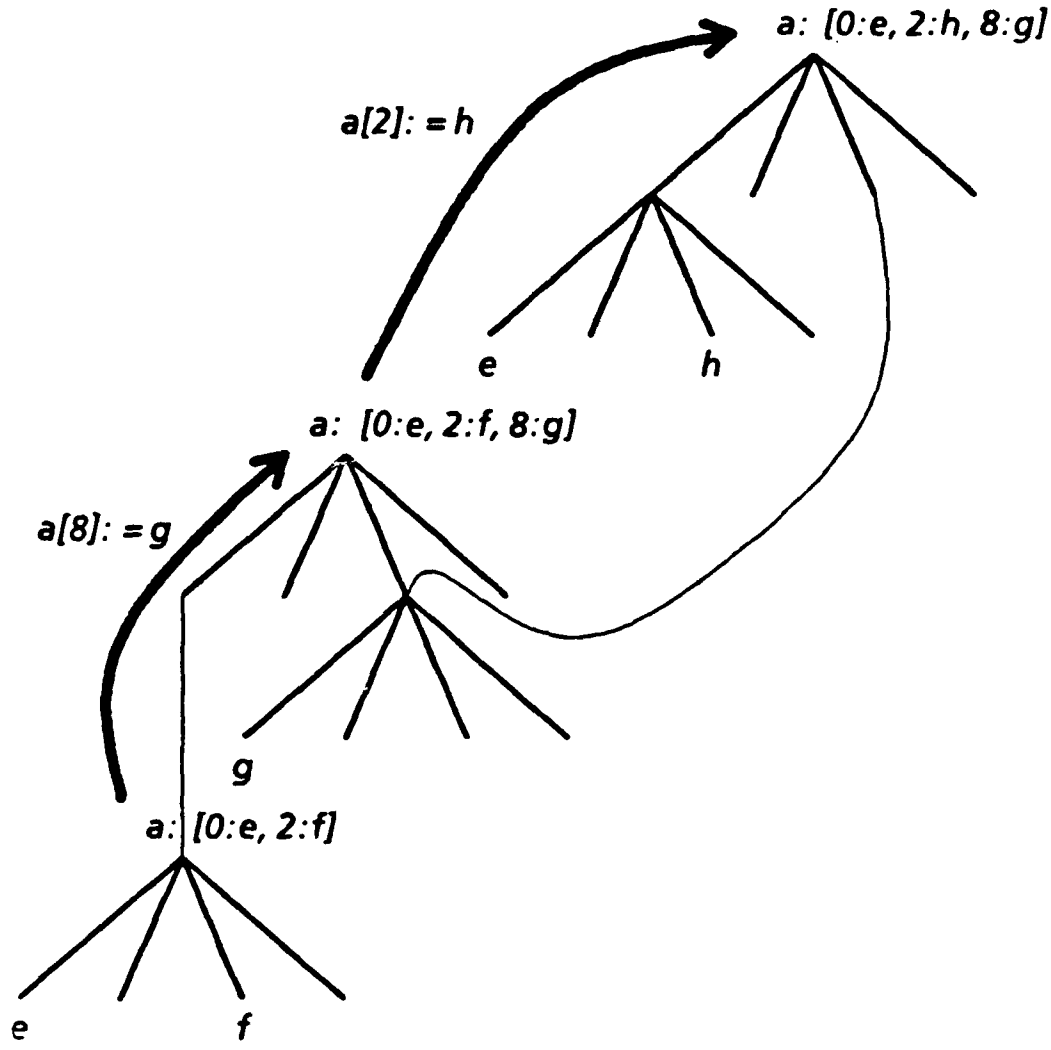


Figure 3.5: Updating Virtual-Copy Arrays

entries or rules.

Derived instances occur in edges that result from the combination of a *left* and a *right parent edge* containing the *left* and *right parent instances* of the derived instance. The *left ancestors* of an instance (edge) are its left parent and that parent's ancestors, and similarly for *right ancestors*. I will assume, for ease of exposition, that a derived instance is always a sub-DAG of the unification of its right parent with a sub-DAG of its left parent. This is the case for most common parsing algorithms, although more general schemes are possible [21].

If the original Boyer-Moore scheme were used directly, the environment for a derived instance would consist of pointers to *left* and *right* parent instances, as well as a list of the updates needed to build the current instance from its parents. As noted before, this method requires a worst-case $O(|d|)$ search to find the updates that result in the current instance.

The present scheme relies on the fact that in the great majority of cases no instance is both the left and the right ancestor of another instance. I shall assume for the moment that this is always the case. In Section 3.9 this restriction will be removed.

It is a simple observation about unification that an update of a node of an instance I is either an update of I's skeleton or of the value (a sub-DAG of another instance) of another update of I. If we iterate this reasoning, it becomes clear that every update is ultimately an update of the skeleton of a base instance ancestor of I. Since we assumed above that no instance could occur more than once in I's derivation, we can therefore conclude that I's environment consists only of updates of nodes in the skeletons of its base instance ancestors. By numbering the base instances of a derivation consecutively, we can then represent an environment by an array of *frames*, each containing all the updates of the skeleton of a given base instance.

Actually, the environment of an instance I will be a *branch environment* containing not only those updates directly relevant to I, but also all those that are relevant to the instances of I's particular branch through the parsing search space.

In the context of a given branch environment, it is then possible to represent a molecule by a pair consisting of a skeleton and the index of a frame in the environment. In particular, this representation can be used for all the values (DAGs) in updates.

More specifically, the frame of a base instance is an array of *update records* indexed by small integers representing the nodes of the instance's skeleton. An update record is either a list of arc bindings for distinct arc labels or a rerouting update. An arc binding is a pair consisting of a label and a molecule (the value of the arc binding). This represents an addition of an arc with that label and that value at the given node. A rerouting update is just a pointer to another molecule; it says that the sub-DAG at that node in the updated DAG is given by that molecule (rather than by whatever was in the initial skeleton).

To see how skeletons and bindings work together to represent a DAG, consider the operation of finding the sub-DAG $d/\langle l_1 \cdots l_m \rangle$ of DAG d. For this purpose, we use a *current skeleton* s and a current frame f, given initially by the skeleton and frame of the molecule representing d. Now assume that the current skeleton s and current frame f correspond to the sub-DAG $d' = d/\langle l_1 \cdots l_{i-1} \rangle$. To find $d/\langle l_1 \cdots l_i \rangle = d'/l_i$, we use the following method:

1. If the top node of s has been rerouted in f to a DAG v, *dereference* d' by setting s and f from v and repeating this step; otherwise

2. If the top node of s has an arc labeled by l_i with value s', the sub-DAG at l_i is given by the moledule (s', f); otherwise

3. If f contains an arc binding labeled l_i for the top node of s, the sub-DAG at l_i is the value of the binding

If none of these steps can be applied, $\langle l_1 \cdots l_i \rangle$ is not a path from the root in d.

The details of the representation are illustrated by the example in Figure 3.6, which shows the passive edges for the chart analysis of the string *ab* according to the sample grammar

$$\begin{array}{lll} S \rightarrow A\ B: & \langle S\ a \rangle = \langle A \rangle \\ & \langle S\ b \rangle = \langle B \rangle \\ & \langle S\ a\ x \rangle = \langle S\ b\ y \rangle \\ A \rightarrow a: & \langle A\ u\ v \rangle = a \\ B \rightarrow b: & \langle B\ u\ v \rangle = b \end{array} \tag{3.3}$$

For the sake of simplicity, only the sub-DAGs corresponding to the explicit equations in these rules are shown (ie., the *cat* DAG arcs and the rule arcs 0, 1,... are omitted). In the figure, the three nonterminal edges (for phrase types S, A and B) are labeled by molecules representing the corresponding DAGs. The skeleton of each of the three molecules comes from the rule used to build the nonterminal. Each molecule points (via a frame index not shown in the figure) to a frame in the branch environment. The frames for the A and B edges contain arc bindings for the top nodes of the respective skeletons whereas the frame for the S edge reroute nodes 1 and 2 of the S rule skeleton to the A and B molecules respectively.

3.7 The Unification Algorithm

I shall now give the unification algorithm for two molecules (DAGs) in the same branch environment.

We can treat a complex DAG d as a partial function from labels to DAGs that maps the label on each arc leaving the top node of the DAG to the DAG at the end of that arc. This allows us to define the following two operations between DAGs:

$$\begin{array}{rcl} d_1 \setminus d_2 & = & \{(l, d) \in d_1 \mid l \notin \mathrm{dom}(d_2)\} \\ d_1 \lhd d_2 & = & \{(l, d) \in d_1 \mid l \in \mathrm{dom}(d_2)\} \end{array}$$

It is clear that $\mathrm{dom}(d_1 \lhd d_2) = \mathrm{dom}(d_2 \lhd d_1)$.

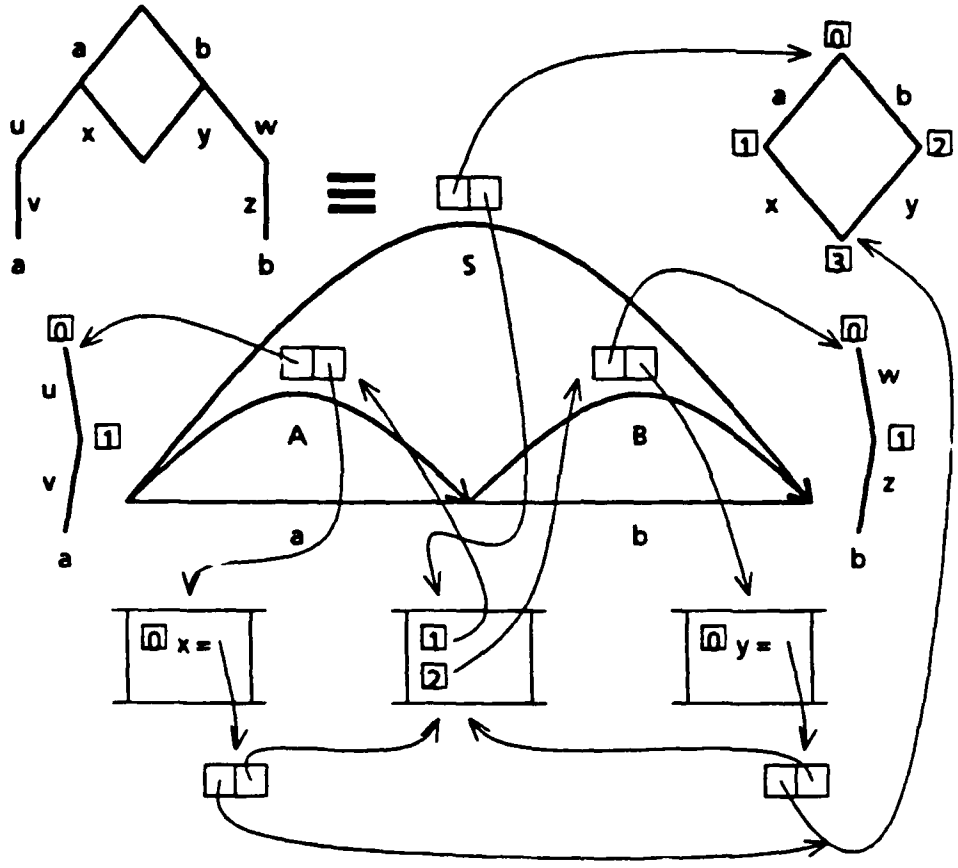


Figure 3.6: Structure-Sharing Chart

We also need the notion of DAG dereferencing introduced in the last section. As a side effect of successive unifications, the top node of a DAG may be rerouted to another DAG whose top node will also end up being rerouted. Dereferencing is the process of following such chains of rerouting pointers to reach a DAG that has not been rerouted.

The unification of DAGs d_1 and d_2 in environment e consists of the following steps:

1. Dereference d_1 and d_2
2. If d_1 and d_2 are identical, the unification is immediately successful
3. If d_1 is a leaf, add to e a rerouting from the top node of d_1 to d_2; otherwise
4. If d_2 is a leaf, add to e a rerouting from the top node of d_2 to d_1; otherwise
5. If d_1 and d_2 are complex DAGs, for each arc $(l, d) \in d_1 \lhd d_2$ unify the DAG d with the DAG d' of the corresponding arc $(l, d') \in d_2 \lhd d_1$. Each of those unifications may add new bindings to e. If this unification of sub-DAGs is successful, all the arcs in $d_1 \setminus d_2$ are are entered in e as arc bindings for the top node of d_2 and finally the top node of d_1 is rerouted to d_2.
6. If none of the conditions above applies, the unification fails.

To determine whether a DAG node is a leaf or complex, both the skeleton and the frame of the corresponding molecule must be examined. For a dereferenced molecule, the set of arcs leaving a node is just the union of the skeleton arcs and the arc bindings for the node. For this to make sense, the skeleton arcs and arc bindings for any molecule node must be disjoint. The interested reader will have no difficulty in proving that this property is preserved by the unification algorithm and therefore all molecules built from skeletons and empty frames by unification will satisfy it.

3.8 Mapping DAGs onto Virtual-Copy Memory

As we saw above, any DAG or set of DAGs constructed by the parser is built from just two kinds of material: (1) frames; (2) pieces of the initial skeletons from rules and lexical entries. The initial skeletons can be represented trivially by host language data structures, as they never change. Frames, though, are always being updated. A new frame is born with the creation of an instance of a rule or lexical entry when the rule or entry is used in some parsing step (uses of the same rule or entry in other steps beget their own frames). A frame is updated when the instance it belongs to participates in a unification.

During parsing, there are in general several possible ways of continuing a derivation. These correspond to alternative ways of updating a branch environment. In abstract terms, on coming to a choice point in the derivation with n possible continuations, $n - 1$ copies of the environment are made, giving n environments — namely, one for each alternative. In fact, the use of virtual-copy arrays for environments and frames renders this copying unnecessary, so each continuation path performs its own updating of its version of the

environment without interfering with the other paths. Thus, all unchanged portions of the environment are shared.

In fact, derivations as such are not explicit in a chart parser. Instead, the instance in each edge has its own branch environment, as described previously. Therefore, when two edges are combined, it is necessary to merge their environments. The cost of this merge operation is at most the same as the worst case cost for unification proper ($O(|d| \log |d|)$). However, in the very common case in which the ranges of frame indices of the two environments do not overlap, the merge cost is only $O(\log |d|)$.

To summarize, we have sharing at two levels: the Boyer-Moore style DAG representation allows derived DAG instances to share input data structures (skeletons), and the virtual-copy array environment representation allows different branches of the search space to share update records.

3.9 The Renaming Problem

In the foregoing discussion of the structure-sharing method, I assumed that the left and right ancestors of a derived instance were disjoint. In fact, it is easy to show that the condition holds whenever the grammar does not allow empty derived edges.

In contrast, it is possible to construct a grammar in which an empty derived edge with DAG D is both a left and a right ancestor of another edge with DAG E. Clearly, the two uses of D as an ancestor of E are mutually independent and the corresponding updates have to be segregated. In other words, we need two copies of the instance D. By analogy with theorem proving, I call this the *renaming* problem.

The current solution is to use *real* copying to turn the empty edge into a skeleton, which is then added to the chart. The new skeleton is then used in the normal fashion to produce multiple instances that are free of mutual interference.

3.10 Implementation

The representation described here has been used in a PATR-II parser implemented in Prolog. Two versions of the parser exist — one using an Earley-style algorithm related to Earley deduction [21], the other using a left-corner algorithm.

Preliminary tests of the left-corner algorithm with structure sharing on various grammars and input have shown parsing times as much as 60% faster (never less, in fact, than 40% faster) than those achieved by the same parsing algorithm with structure copying.

Acknowledgments

Thanks are due to Stuart Shieber, Lauri Karttunen, and Ray Perrault for their comments on earlier presentations of this material.

References

[1] Boyer, R.S. and J S. Moore. The sharing of structure in theorem-proving programs. In B. Meltzer and D. Michie (eds.). *Machine Intelligence 7*. Edinburgh University Press, Edinburgh, Scotland, 1972, 101–116.

[2] Kaplan, R. and J. Bresnan. Lexical-functional grammar: a formal system for grammatic representation. In J. Bresnan (ed.). *The Mental Representation of Grammatical Relations*. MIT Press, Cambridge, Massachusetts, 1983.

[3] Kay, M. Algorithm schemata and data structures in syntactic processing. Technical Report, Xerox Palo Alto Research Center, Palo Alto, California, 1983.

[4] Kay, M. Unification grammar. Technical Report, Xerox Palo Alto Research Center, Palo Alto, California, 1983.

[5] Pereira, F.C.N. and S.M. Shieber. The semantics of grammar formalisms seen as computer languages. *Proceedings of the Tenth International Conference on Computational Linguistics*, Stanford University, Stanford, California, July 1984.

[6] Pereira, F.C.N. and D.H.D. Warren. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence 13* (1980), 231–278.

[7] Pereira, F.C.N. and D.H.D. Warren. *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1983.

[8] Sheiber, S.M. The design of a computer language for linguistic interpretation. *Proceedings of the Tenth International Conference on Computational Linguistics*, Stanford University, Stanford, California, July 1984.

[9] Warren, D.H.D. *Applied Logic—Its Use and Implementation as a Programming Tool*. Ph.D. dissertation, University of Edinburgh, Edinburgh, Scotland, 1977. Reprinted at Technical Note 290, Artificial Intelligence Center, SRI International, Menlo Park, California.

[10] Warren, D.H.D. Logarithmic access arrays for Prolog. Unpublished program, 1983.

Chapter 4

Structure Sharing with Binary Trees

This chapter was written by Lauri Karttunen and Martin Kay[1].

Many current interfaces for natural language represent syntactic and semantic information in the form of directed graphs where attributes correspond to vectors and values to nodes. There is a simple correspondence between such graphs and the matrix notation linguists traditionally use for feature sets.

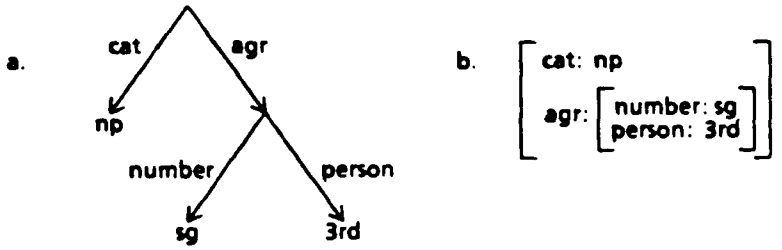


Figure 1

The standard operation for working with such graphs is unification. The unification operation succedes only on a pair of compatible graphs, and its result is a graph containing the information in both contributors. When a parser applies a syntactic rule, it unifies selected features of input constituents to check constraints and to build a representation for the output constituent.

[1]Xerox Palo Alto Research Center and the Center For the Study of Language and Information, Stanford University

4.1 Problem: Proliferation of Copies

When words are combined to form phrases, unification is not applied to lexical representations directly because it would result in the lexicon being changed. When a word is encountered in a text, a copy is made of its entry, and unification is applied to the copied graph, not the original one. In fact, unification in a typical parser is always preceded by a copying operation. Because of nondeterminism in parsing, it is, in general, necessary to preserve every representation that gets built. The same graph may be needed again when the parser comes back to pursue some yet unexplored option. Our experience suggests that the amount of computational effort that goes into producing these copies is much greater than the cost of unification itself. It accounts for a significant amount of the total parsing time. In a sense, most of the copying effort is wasted. Unifications that fail typically fail for a simple reason. If it were known in advance what aspects of structures are relevant in a particular case, some effort could be saved by first considering only the crucial features of the input.

4.2 Solution: Structure Sharing

We lay out one strategy that has turned out to be very useful in eliminating much of the wasted effort. Our version of the basic idea is due to Martin Kay. It has been implemented in slightly different ways by Kay in Interlisp-D and by Lauri Karttunen in Zeta Lisp. The basic idea is to minimize copying by allowing graphs share common parts of their structure. This version of structure sharing is based on four related ideas:

- Binary trees as a storage device for feature graphs
- "Lazy" copying
- Relative indexing of nodes in the tree
- Strategy for keeping storage trees as balanced as possible

4.3 Binary Trees

Our structure-sharing scheme depends on represented feature sets as binary trees. A tree consists of cells that have a content field and two pointers which, if not empty, point to a left and a right cell respectively. For example, the content of the feature set and the corresponding directed graph in Figure 1 can be distributed over the cells of a binary tree

in the following way.

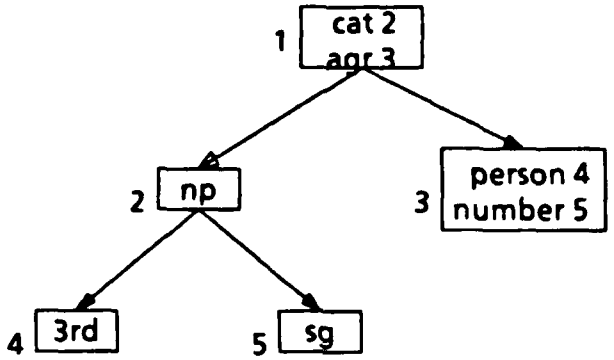


Figure 2

The index of the top node is 1; the two cells below have indices 2 and 3. In general, a node whose index is n may be the parent of cells indexed $2n$ and $2n + 1$. Each cell contains either an atomic value or a set of pairs that associate attribute names with indices of cells where their value is stored. The assignment of values to storage cells is arbitrary; it doesn't matter which cell stores which value. Here, cell 1 contains the information that the value of the attribute cat is found in cell 2 and that of agr in cell 3. This is a slight simplification. As we shall shortly see, when the value in a cell involves a reference to another cell, that reference is encoded as a relative index. The method of locating the cell that corresponds to a given index takes advantage of the fact that the tree branches in a binary fashion. The path to a node can be read off from the binary representation of its index by starting after the first 1 in this number and taking 0 to be a signal for a left turn and 1 as a mark for a right turn. For example, starting at node 1, node 5 is reached by first going down a left branch and then a right branch. This sequence of turns corresponds to the digits 01. Prefixed with 1, this is the same as the binary representation of 5, namely 101. The same holds for all indices. Thus the path to node 9 (binary 1001) would be LEFT-LEFT-RIGHT as signalled by the last three digits following the initial 1 in the binary numeral (see Figure 6).

4.4 Lazy Copying

The most important advantage is that the scheme minimizes the amount of copying that has to be done. In general, when a graph is copied, we duplicate only The operation that replaces copying in this scheme starts by duplicating the topmost node of the tree that contains it. The rest of the structure remains the same. Other nodes are modified only if and when destructive changes are about to happen. For example, assume that we need another copy of the graph stored in the tree in Figure 2. This can be obtained by producing a tree which has a different root node, but shares the rest of the structure with its original.

In order to keep track of which tree actually owns a given node, each node carries a numeral tag that indicates its parentage. The relationship between the original tree (generation 0) and its copy (generation 1) is illustrated in Figure 3 where the generation is separated from the index of a node by a colon.

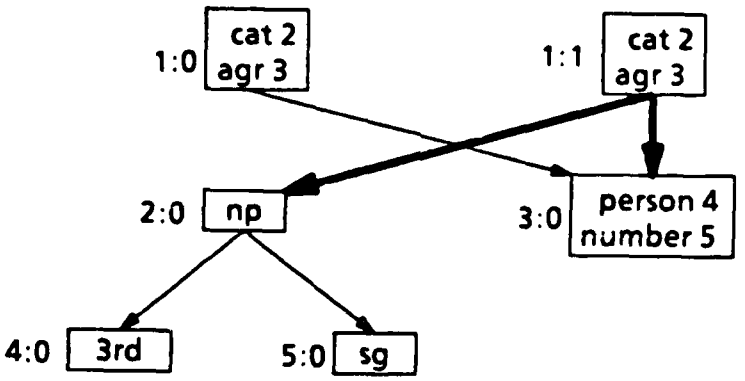


Figure 3

If the node that we want to copy is not the topmost node of a tree, we need to duplicate the nodes along the branch leading to it.

When a tree headed by the copied node has to be changed, we use the generation tags to minimize the creation of new structure. In general, all and only the nodes on the branch that lead to the site of a destructive change or addition need to belong to the same generation as the top node of the tree. The rest of the structure can consist of old nodes. For example, suppose we add a new feature, say [gender: fem] to the value of agr in Figure 3 to yield the feature set in Figure 4.

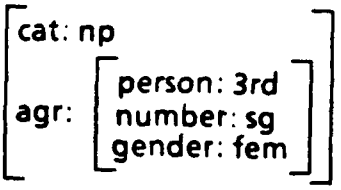


Figure 4

Furthermore, suppose that we want the change to affect only the copy but not the original feature set. In terms of the trees that we have constructed for the example in Figure 3, this involves adding one new cell to the copied structure to hold the value fem,

and changing the content of cell 3 by adding the new feature to it.

The modified copy and its relation to the original is shown in Figure 5 . Note that one half of the structure is shared. The copy contains only three new nodes.

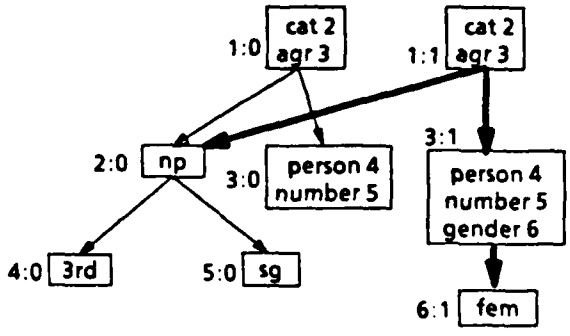


Figure 5

From the point of view of a process that only needs to find or print out the value of particular features, it makes no difference that the nodes containing the values belong to several trees as long as there is no confusion about the structure.

4.5 Relative Addressing

Accessing an arbitrary cell in a binary tree consumes time in proportion to the logarithm of the size of the structure, assuming that cells are reached by starting at the top node and using the index of the target node as an address. Another method is to use relative addressing. Relative addresses encode the shortest path between two nodes in the tree regardless of where they are are. For example, if we are at node 9 in Figure 6.a below and need to reach node 11, it is easy to see that it is not necessary to go all the way up to node 1 and then partially retrace the same path in looking up node 11. Instead, one can stop going upward at the lowest common ancestor, node 2, of nodes 9 and 11 and go down from

there.

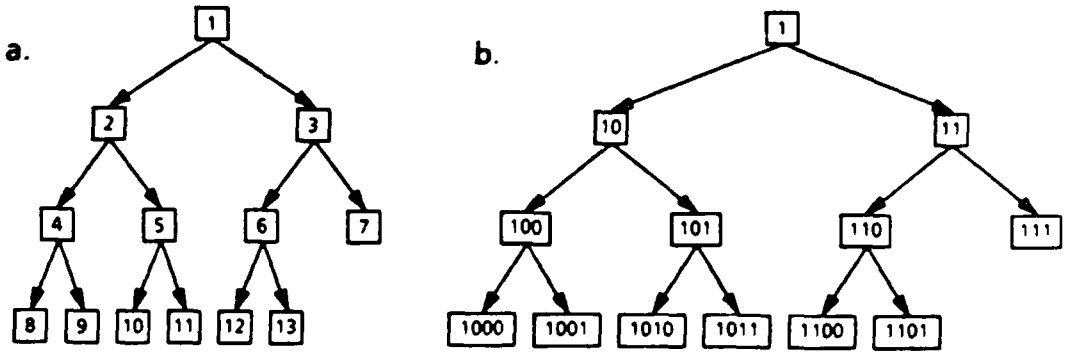


Figure 6

With respect to node 2, node 11 is in the same position as 7 is with respect 1. Thus the relative address of cell 11 counted from 9 is 2,7—"two nodes up, then down as if going to node 7". In general, relative addresses are of the form ⟨up,down⟩ where ⟨up⟩ is the number of links to the lowest common ancestor of the origin and ⟨down⟩ is the relative index of the target node with respect to it. Sometimes we can just go up or down on the same branch: for example, the relative address of cell 10 seen from node 2 is simply 0,6: the path from 8 or 9 to 4 is 1,1. As one might expect, it is easy to see these relationships if we think of node indices in their binary representation (see Figure 6.b). The lowest common ancestor 2 (binary 10) is designated by the longest common initial substring of 9 (binary 1001) and 11 (binary 1011). The relative index of 11, with respect to, 7 (binary 111), is the rest of its index with 1 prefixed to the front.

In terms of number of links traversed, relative addresses have no statistical advantage over the simpler method of always starting from the top. However, they have one important property that is essential for our purposes: relative addresses remain valid even when trees are embedded in other trees; absolute indices would have to be recalculated. Figure 7 is a

recoding of Figure 5 using relative addresses.

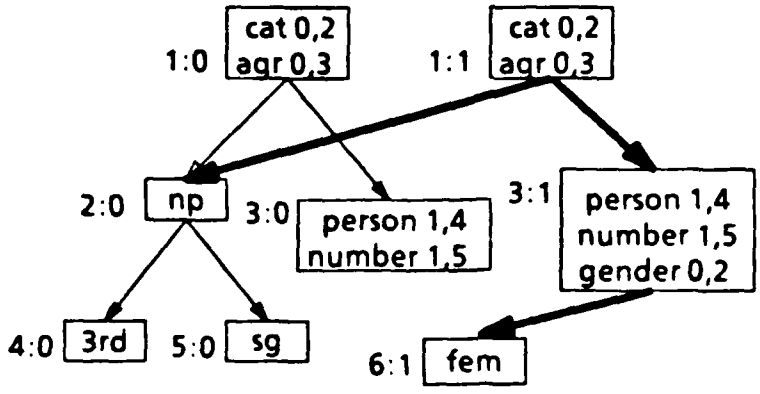


Figure 7

4.6 Keeping Trees Balanced

When two feature matrices are unified, the binary trees corresponding to them have to be combined to form a single tree. New attributes are added to some of the nodes; other nodes become "pointer nodes," i.e., their only content is the relative address of some other node where the real content is stored. As long as we keep adding nodes to one tree, it is a simple matter to keep the tree maximally balanced. At any given time, only the growing fringe of the tree can be incompletely filled. When two trees need to be combined, it would, of course, be possible to add all the cells from one tree in a balanced fashion to the other one but that would defeat the very purpose of using binary trees because it would mean having to copy almost all of the structure. The only alternative is to embed one of the trees in the other one. The resulting tree will not be a balanced one; some of the branches are much longer than others. Consequently, the average time needed to look up a value is bound to be worse than in a balanced tree. For example, suppose that we want to unify a copy of the feature set in Figure 1b, represented as in Figure 2 but with relative addressing, with a

copy of the feature set in Figure 8.

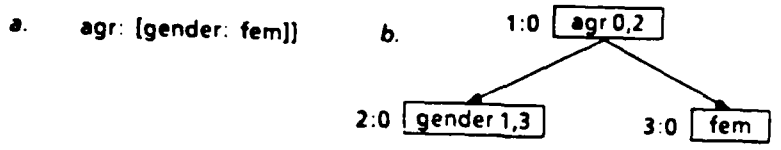


Figure 8

The resulting feature set and structure are shown in Figure 9.

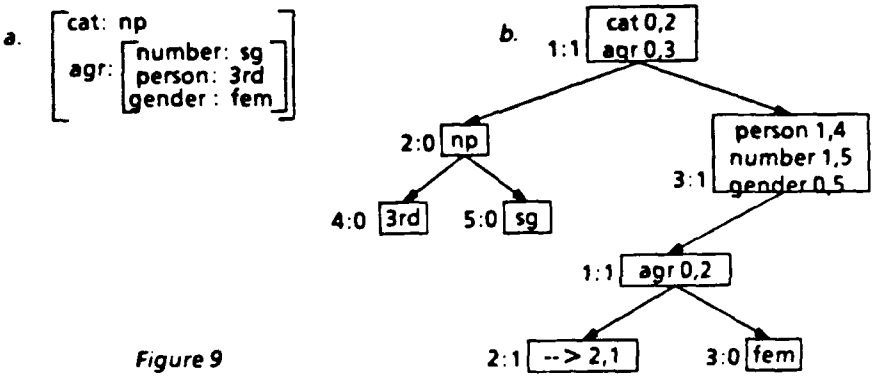


Figure 9

Although the feature set in Figure 9a is the same as the one represented by the right half of Figure 7, the structure in Figure 9b is more complicated because it is derived by unifying copies of two separate trees, not by simply adding more features to a tree, as in Figure 7. In 9b, a copy of 8b has been embedded as node 6 of the host tree. The original indices of both trees remain unchanged. Because all the addresses are relative; no harm comes from the fact that indices in the embedded tree no longer correspond to the true location of the nodes. Absolute indices are not used as addresses because they change when a tree is embedded. The symbol --> in node 2 of the lower tree indicates that the original content of this node—gender 1,3—has been replaced by the address of the cell that it was unified with, namely cell 3 in the host tree. In the case at hand, it matters very little which of the two trees becomes the host for the other. The resulting tree is about as much out of balance either way. However, when a sequence of unifications is performed, differences can be very significant. For example, if A, B, and C are unified with one another, it can make a great deal of difference, which of the two alternative shapes in Figure 10 is produced as

the final result.

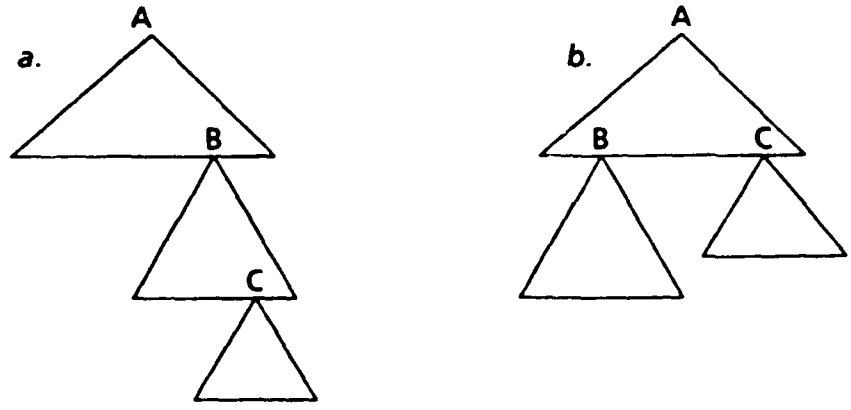


Figure 10

When a choice has to be made as to which of the two trees to embed in the other, it is important to minimize the length of the longest path in the resulting tree. To do this at all eificiently requires addtitional infornation to be stored with each node. According to one simple scheme, this is simply the length of the shortest path from the node down to a node with a free left or right pointer. Using this, it is a simple matter to find the shallowest place in a tree at which to embed another one. If the length of the longet path is also stored, it is also easy to determine which choice of host will give rise to the shallowest combined tree. Another problem which needs careful attention concerns generation markers. If a pair of trees to be unified have independent histories, their generation markers will presumably be incommensurable and those of an embedded tree will therfore not be valide in the host. Various solutions are possible for this problem. The most straightforward is relate the histories of all trees at least to the extent of drawing generation markers from a global pool. In Lisp, for example, the simplest thing is to let them be CONS cells.

4.7 Conclusion

We will conclude by comparing our method of structure sharing with two others that we know of: R. Cohen's immutable arrays and the idea discussed in the previous chapter by Fernando Pereira. The three alternatives involve different trade-offs along the space/time continuum. The choice between them will depend on the particular application they are intended for.

Acknowledgments

Thanks are due to Fernando Pereira and Stuart Shieber for their comments on earlier presentations of this material.

Chapter 5

Using Restriction to Extend Parsing Algorithms for Complex-Feature-Based Formalisms

This chapter was written by Stuart Shieber.

5.1 Introduction

Grammar formalisms based on the encoding of grammatical information in complex-valued feature systems enjoy some currency both in linguistics and natural-language-processing research. Such formalisms can be thought of by analogy to context-free grammars as generalizing the notion of nonterminal symbol from a finite domain of atomic elements to a possibly infinite domain of directed graph structures of a certain sort. Many of the surface-based grammatical formalisms explicitly defined or presupposed in linguistics can be characterized in this way—e.g., lexical-functional grammar (LFG) [4], generalized phrase structure grammar (GPSG) [4], even categorial systems such as Montague grammar [18] and Ades/Steedman grammar [1]—as can several of the grammar formalisms being used in natural-language processing research—e.g., definite clause grammar (DCG) [7], and PATR-II [12].

Unfortunately, in moving to an infinite nonterminal domain, standard methods of parsing may no longer be applicable to the formalism. For instance, the application of techniques for preprocessing of grammars in order to gain efficiency may fail to terminate, as in left-corner and LR algorithms. Algorithms performing top-down prediction (e.g. top-down backtrack parsing, Earley's algorithm) may not terminate at parse time. Implementing backtracking regimens—useful for instance for generating parses in some particular order, say, in order of syntactic preference—is in general difficult when LR-style and top-down backtrack techniques are eliminated.

In this work, we discuss a solution to the problem of extending parsing algorithms to formalisms with possibly infinite nonterminal domains, a solution based on an operation we call restriction. In Section 5.2, we summarize traditional proposals for solutions and problems inherent in them and propose an alternative approach to a solution using restriction. In Section 5.3, we present some technical background including a brief description of the PATR-II formalism—which is used as the formalism interpreted by the parsing algorithms—and a formal definition of restriction for PATR-II's nonterminal domain. In Section 5.4, we develop a correct, complete and terminating extension of Earley's algorithm for the PATR-II formalism using the restriction notion. Readers uninterested in the technical details of the extensions may want to skip these latter two sections, referring instead to Section 5.4.1 for an informal overview of the algorithms. Finally, in Section 5.5, we discuss applications of the particular algorithm and the restriction technique in general.

5.2 Traditional Solutions and an Alternative Approach

Problems with efficiently parsing formalisms based on potentially infinite nonterminal domains have manifested themselves in many different ways. Traditional solutions have involved limiting in some way the class of grammars that can be parsed.

5.2.1 Limiting the Formalism

The limitations can be applied to the formalism by, for instance, adding a context-free "backbone." If we require that a context-free subgrammar be implicit in every grammar, the subgrammar can be used for parsing and the rest of the grammar used as a filter during or after parsing. This solution has been recommended for functional unification grammars (FUG) by Martin Kay [6]; its legacy can be seen in the context-free skeleton of LFG, and the Hewlett-Packard GPSG system [3], and in the *cat* feature requirement in PATR-II that is described below.

However, several problems inhere in this solution of mandating a context-free backbone. First, the move from context-free to complex-feature-based formalisms was motivated by the desire to structure the notion of nonterminal. Many analyses take advantage of this by eliminating mention of major category information from particular rules[1] or by structuring the major category itself (say into binary N and V features plus a bar level-feature as in $\overline{X}$-based theories). Forcing the primacy and atomicity of major category defeats part of the purpose of structured category systems.

Second, and perhaps more critically, because only certain of the information in a rule is used to guide the parse, say major category information, only such information can be used to filter spurious hypotheses by top-down filtering. Note that this problem occurs even if filtering by the rule information is used to eliminate at the earliest possible time constituents and partial constituents proposed during parsing (as is the case in the PATR-II

[1]See, for instance, the coordination and copular "be" analyses from GPSG [4], the nested VP analysis used in some PATR-II grammars [15], or almost all categorial analyses, in which general rules of combination play the role of specific phrase-structure rules.

implementation and the Earley algorithm given below; cf. the Xerox LFG system). Thus, if information about subcategorization is left out of the category information in the context-free skeleton, it cannot be used to eliminate prediction edges. For example, if we find a verb that subcategorizes for a noun phrase, but the grammar rules allow postverbal NPs, PPs, $\overline{S}$s, VPs, and so forth, the parser will have no way to eliminate the building of edges corresponding to these categories. Only when such edges attempt to join with the V will the inconsistency be found. Similarly, if information about filler-gap dependencies is kept extrinsic to the category information, as in a slash category in GPSG or an LFG annotation concerning a matching constituent for a $\Uparrow$ specification, there will be no way to keep from hypothesizing gaps at any given vertex. This "gap-proliferation" problem has plagued many attempts at building parsers for grammar formalisms in this style.

In fact, by making these stringent requirements on what information is used to guide parsing, we have to a certain extent thrown the baby out with the bathwater. These formalisms were intended to free us from the tyranny of atomic nonterminal symbols, but for good performance, we are forced toward analyses putting more and more information in an atomic category feature. An example of this phenomenon can be seen in the author's paper on LR syntactic preference parsing [14]. Because the LALR table building algorithm does not in general terminate for complex-feature-based grammar formalisms, the grammar used in that paper was a simple context-free grammar with subcategorization and gap information placed in the atomic nonterminal symbol.

5.2.2 Limiting Grammars and Parsers

On the other hand, the grammar formalism can be left unchanged, but particular grammars developed that happen not to succumb to the problems inherent in the general parsing problem for the formalism. The solution mentioned above of placing more information in the category symbol falls into this class. Unpublished work by Kent Wittenburg and by Robin Cooper has attempted to solve the gap proliferation problem using special grammars.

In building a general tool for grammar testing and debugging, however, we would like to commit as little as possible to a particular grammar or style of grammar.[2] Furthermore, the grammar designer should not be held down in building an analysis by limitations of the algorithms. Thus a solution requiring careful crafting of grammars is inadequate.

Finally, specialized parsing algorithms can be designed that make use of information about the particular grammar being parsed to eliminate spurious edges or hypotheses. Rather than using a general parsing algorithm on a limited formalism, Ford, Bresnan, and Kaplan [2] chose a specialized algorithm working on grammars in the full LFG formalism to model syntactic preferences. Current work at Hewlett-Packard on parsing recent variants of GPSG seems to take this line as well.

Again, we feel that the separation of burden is inappropriate in such an attack, especially in a grammar-development context. Coupling the grammar design and parser design problems in this way leads to the linguistic and technological problems becoming inherently mixed, magnifying the difficulty of writing an adequate grammar/parser system.

[2] See [13] for further discussion of this matter.

5.2.3 An Alternative: Using Restriction

Instead, we would like a parsing algorithm that placed no restraints on the grammars it could handle as long as they could be expressed within the intended formalism. Still, the algorithm should take advantage of that part of the arbitrarily large amount of information in the complex-feature structures that is *significant* for guiding parsing with the particular grammar. One of the aforementioned solutions is to require the grammar writer to put all such significant information in a special atomic symbol—i.e., mandate a context-free backbone. Another is to use *all* of the feature structure information—but this method, as we shall see, inevitably leads to nonterminating algorithms.

A compromise is to parameterize the parsing algorithm by a small amount of grammar-dependent information that tells the algorithm *which* of the information in the feature structures is significant for guiding the parse. That is, the parameter determines how to split up the infinite nonterminal domain into a finite set of equivalence classes that can be used for parsing. By doing so, we have an optimal compromise: Whatever part of the feature structure is significant we distinguish in the equivalence classes by setting the parameter appropriately, so the information is used in parsing. But because there are only a finite number of equivalence classes, parsing algorithms guided in this way will terminate.

The technique we use to form equivalence classes is *restriction*, which involves taking a quotient of the domain with respect to a *restrictor*. The restrictor thus serves as the sole repository of grammar-dependent information in the algorithm. By tuning the restrictor, the set of equivalence classes engendered can be changed, making the algorithm more or less efficient at guiding the parse. But independent of the restrictor, the algorithm will be correct, since it is still doing parsing over a finite domain of "nonterminals," namely, the elements of the restricted domain.

This idea can be applied to solve many of the problems engendered by infinite nonterminal domains, allowing preprocessing of grammars as required by LR and LC algorithms, allowing top-down filtering or prediction as in Earley and top-down backtrack parsing, guaranteeing termination, etc.

5.3 Technical Preliminaries

Before discussing the use of restriction in parsing algorithms, we present some technical details, including a brief introduction to the PATR-II grammar formalism, which will serve as the grammatical formalism that the presented algorithms will interpret. PATR-II is a simple grammar formalism that can serve as the least common denominator of many of the complex-feature-based and unification-based formalisms prevalent in linguistics and computational linguistics. As such it provides a good testbed for describing algorithms for complex-feature-based formalisms.

5.3.1 The PATR-II Nonterminal Domain

The PATR-II nonterminal domain is a lattice of directed, acyclic, graph structures (DAGs).[3] DAGs can be thought of as similar to the reentrant f-structures of LFG or functional structures of FUG, and we will use the bracketed notation associated with these formalisms for them. For example, the following is a DAG (D_0) in this notation, with reentrancy indicated with coindexing boxes:

$$\left[\begin{array}{ll} a: & \left[\, b: \;\; c \,\right] \\ d: & \left[\begin{array}{ll} e: & \boxed{1}\left[\, f: \;\; \left[\, g: \;\; h \,\right] \,\right] \\ i: & \left[\, j: \;\; \boxed{1} \,\right] \\ k: & l \end{array}\right] \end{array}\right]$$

DAGs come in two varieties, *complex* (like the one above) and *atomic* (like the DAGs h and c in the example). Complex DAGs can be viewed as partial functions from labels to DAG values, and the notation $D(l)$ will therefore denote the value associated with the label l in the DAG D. In the same spirit, we can refer to the domain of a DAG ($dom(D)$). A DAG with an empty domain is often called an *empty* DAG or *variable*. A *path* in a DAG is a sequence of label names (notated, e.g., $\langle d\ e\ f \rangle$), which can be used to pick out a particular subpart of the DAG by repeated application (in this case, the DAG $[g : h]$). We will extend the notation $D(p)$ in the obvious way to include the sub-DAG of D picked out by a path p. We will also occasionally use the square brackets as the DAG constructor function, so that $[f : D]$ where D is an expression denoting a DAG will denote the DAG whose f feature has value D.

5.3.2 Subsumption and Unification

There is a natural lattice structure for DAGs based on *subsumption*—an ordering on DAGs that roughly corresponds to the compatibility and relative specificity of information contained in the DAGs. Intuitively viewed, a DAG D subsumes a DAG D' (notated $D \sqsubseteq D'$) if D contains a subset of the information in (i.e., is more general than) D'.

Thus variables subsume all other DAGs, atomic or complex, because as the trivial case, they contain no information at all. A complex DAG D subsumes a complex DAG D' if and only if $D(l) \sqsubseteq D'(l)$ for all $l \in dom(D)$ and $D'(P) = D'(q)$ for all paths p and q such that $D(p) = D(q)$. An atomic DAG neither subsumes nor is subsumed by any different atomic DAG.

For instance, the following subsumption relations hold:

$$[\ ] \sqsubseteq [d : e] \sqsubseteq \left[\begin{array}{ll} a: & [b : c] \\ e: & f \end{array}\right] \sqsubseteq \left[\begin{array}{ll} a: & \boxed{1}[b : c] \\ d: & \boxed{1} \\ e: & f \end{array}\right] .$$

[3] The reader is referred to earlier works [15,10] for more detailed discussions of DAG structures.

Finally, given two DAGs D' and D'', the *unification* of the DAGs is the most general DAG D such that $D' \sqsubseteq D$ and $D'' \sqsubseteq D$. We notate this $D = D' \sqcup D''$.

The following examples illustrate the notion of unification:

$$\left[\begin{array}{ll} a: & [b:c] \end{array}\right] \sqcup \left[\begin{array}{ll} d: & e \end{array}\right] = \left[\begin{array}{ll} a: & [b:c] \\ d: & e \end{array}\right]$$

$$\left[\begin{array}{ll} a: & [b:c] \end{array}\right] \sqcup \left[\begin{array}{ll} a: & \boxed{1}[] \\ d: & \boxed{1} \end{array}\right] = \left[\begin{array}{ll} a: & \boxed{1}[b:c] \\ d: & \boxed{1} \end{array}\right] \quad .$$

The unification of two DAGs is not always well-defined. In the cases where no unification exists, the unification is said to *fail*. For example the following pair of DAGs fail to unify with each other:

$$\left[\begin{array}{ll} a: & \boxed{1}[] \\ d: & \boxed{1} \end{array}\right] \sqcup \left[\begin{array}{ll} a: & [b:c] \\ d: & [b:d] \end{array}\right] = \mathit{fail} \quad .$$

5.3.3 Restriction in the PATR-II Nonterminal Domain

Now, consider the notion of *restriction* of a DAG, using the term almost in its technical sense of restricting the domain of a function. By viewing DAGs as partial functions from labels to DAG values, we can envision a process of restricting the domain of this function to a given set of labels. Extending this process recursively to every level of the DAG, we have the concept of restriction used below. Given a finite specification Φ (called a restrictor) of what the allowable domain at each node of a DAG is, we can define a functional, $\restriction$, that yields the DAG restricted by the given restrictor.

Formally, we define restriction as follows. Given a relation Φ between paths and labels, and a DAG D, we define $D\restriction\Phi$ to be the most specific DAG $D' \sqsubseteq D$ such that for every path p either $D'(p)$ is undefined, or $D'(p)$ is atomic, or for every $l \in dom(D'(p))$, $p\Phi l$. That is, every path in the restricted DAG is either undefined, atomic, or *specifically allowed* by the restrictor.

The restriction process can be viewed as putting DAGs into equivalence classes, each equivalence class being the largest set of DAGs that all are restricted to the same DAG (which we will call its *canonical member*). It follows from the definition that in general $D\restriction\Phi \sqsubseteq D$. Finally, if we disallow infinite relations as restrictors (i.e., restrictors must not allow values for an infinite number of distinct paths) as we will do for the remainder of the discussion, we are guaranteed to have only a finite number of equivalence classes.

Actually, in the sequel we will use a particularly simple subclass of restrictors that are generable from sets of paths. Given a set of paths s, we can define Φ such that $p\Phi l$ if and only if p is a prefix of some $p' \in s$. Such restrictors can be understood as "throwing away" all values not lying on one of the given paths. This subclass of restrictors is sufficient for most applications. However, the algorithms that we will present apply to the general class as well.

Using our previous example, consider a restrictor Φ_0 generated from the set of paths $\{\langle a\ b\rangle, \langle d\ e\ f\rangle, \langle d\ i\ j\ f\rangle\}$. That is, $p\Phi_0 l$ for all p in the listed paths and all their prefixes. Then given the previous DAG D_0, $D_0 \restriction \Phi_0$ is

$$\left[\begin{array}{ll} a: & \left[\begin{array}{ll} b: & c \end{array}\right] \\ d: & \left[\begin{array}{ll} e: & \boxed{1}\left[\begin{array}{ll} f: & [] \end{array}\right] \\ i: & \left[\begin{array}{ll} j: & \boxed{1} \end{array}\right] \end{array}\right] \end{array}\right] .$$

Restriction has thrown away all the information except the direct values of $\langle a\ b\rangle$, $\langle d\ e\ f\rangle$, and $\langle d\ i\ j\ f\rangle$. (Note however that because the values for paths such as $\langle d\ e\ f\ g\rangle$ were thrown away, $(D_0 \restriction \Phi_0)(\langle d\ e\ f\rangle)$ is a variable.)

5.3.4 PATR-II Grammar Rules

PATR-II rules describe how to combine a sequence of constituents, $X_1, \ldots, X_n$ to form a constituent X_0, stating mutual constraints on the DAGs associated with the $n+1$ constituents as unifications of various parts of the DAGs. For instance, we might have the following rule:

$X_0 \rightarrow X_1 X_2$:
$\langle X_0\ cat\rangle = S$
$\langle X_1\ cat\rangle = NP$
$\langle X_2\ cat\rangle = VP$
$\langle X_1\ agreement\rangle = \langle X_2\ agreement\rangle$.

By notational convention, we can eliminate unifications for the special feature *cat* (the atomic major category feature) recording this information implicitly by using it in the "name" of the constituent, e.g.,

$S \rightarrow NP\ VP$:
$\langle NP\ agreement\rangle = \langle VP\ agreement\rangle$.

If we *require* that this notational convention always be used (in so doing, guaranteeing that each constituent have an atomic major category associated with it), we have thereby mandated a context-free backbone to the grammar, and can then use standard context-free parsing algorithms to parse sentences relative to grammars in this formalism. Limiting to a context-free-based PATR-II is the solution that previous implementations have incorporated.

Before proceeding to describe parsing such a context-free-based PATR-II, we make one more purely notational change. Rather than associating with each grammar rule a set of

unifications, we instead associate a DAG that incorporates all of those unifications implicitly, i.e., a rule is associated with a DAG D_r such that for all unifications of the form $p = q$ in the rule, $D_r(p) = D_r(q)$. Similarly, unifications of the form $p = a$ where a is atomic would require that $D_r(p) = a$. For the rule mentioned above, such a DAG would be

$$\left[\begin{array}{ll} X_0: & \left[\begin{array}{ll} cat: & S \end{array}\right] \\ X_1: & \left[\begin{array}{ll} cat: & NP \\ agreement: & \boxed{1}[] \end{array}\right] \\ X_2: & \left[\begin{array}{ll} cat: & VP \\ agreement: & \boxed{1} \end{array}\right] \end{array}\right]$$

Thus a rule can be thought of as an ordered pair $\langle P, D \rangle$ where P is a production of the form $X_0 \rightarrow X_1 \cdots X_n$ and D is a DAG with top-level features $X_0, \ldots, X_n$ and with atomic values for the *cat* feature of each of the top-level sub-DAGs. The two notational conventions—using sets of unifications instead of DAGs, and putting the *cat* feature information implicitly in the names of the constituents—allow us to write rules in the more compact and familiar format above, rather than this final cumbersome way presupposed by the algorithm.

5.4 Using Restriction to Extend Earley's Algorithm for PATR-II

We now develop a concrete example of the use of restriction in parsing by extending Earley's algorithm to parse grammars in the PATR-II formalism just presented.

5.4.1 An Overview of the Algorithms

Earley's algorithm is a bottom-up parsing algorithm that uses top-down prediction to hypothesize the starting points of possible constituents. Typically, the prediction step determines which *categories* of constituent can start at a given point in a sentence. But when most of the information is not in an atomic category symbol, such prediction is relatively useless and many types of constituents are predicted that could never be involved in a completed parse. This standard Earley's algorithm is presented in Section 5.4.2.

By extending the algorithm so that the predictor step determines which *DAGs* can start at a given point, we can use the information in the features to be more precise in the predictions and eliminate many hypotheses. However, because there are a potentially infinite number of such feature structures, the predictor step may never terminate. This extended Earley's algorithm is presented in Section 5.4.3.

We compromise by having the predictor step determine which *restricted DAGs* can start at a given point. If the restrictor is chosen appropriately, this can be as constraining as predicting on the basis of the whole feature structure, yet prediction is guaranteed to terminate because the domain of restricted feature structures is finite. This final extension of Earley's algorithm is presented in Section 5.4.4.

5.4.2 Parsing a Context-Free-Based PATR-II

We start with the Earley algorithm for context-free-based PATR-II on which the other algorithms are based. The algorithm is described in a chart-parsing incarnation, vertices numbered from 0 to n for an n-word sentence $w_1 \cdots w_n$. An item of the form $[h, i, A \rightarrow \alpha.\beta, D]$ designates an edge in the chart from vertex h to i with dotted rule $A \rightarrow \alpha.\beta$ and DAG D.

The chart is initialized with an edge $[0, 0, X_0 \rightarrow .\alpha, D]$ for each rule $\langle X_0 \rightarrow \alpha, D \rangle$ where $D(\langle X_0\ cat \rangle) = S$.

For each vertex i do the following steps until no more items can be added:

predictor step: For each item ending at i of the form $[h, i, X_0 \rightarrow \alpha.X_j\beta, D]$ and each rule of the form $\langle X_0 \rightarrow \gamma, E \rangle$ such that $E(\langle X_0\ cat \rangle) = D(\langle X_j\ cat \rangle)$, add an edge of the form $[i, i, X_0 \rightarrow .\gamma, E]$ if this edge is not subsumed by another edge.

Informally, this involves *predicting top-down all rules whose left-hand-side category matches the category of some constituent being looked for.*

Completer step: For each item of the form $[h, i, X_0 \rightarrow \alpha., D]$ and each item of the form $[g, h, X_0 \rightarrow \beta.X_j\gamma, E]$ add the item $[g, i, X_0 \rightarrow \beta X_j.\gamma, E \sqcup [X_j : D(X_0)]]$ if the unification succeeds[4] and this edge is not subsumed by another edge.[5]

Informally, this involves *forming a new partial phrase whenever the category of a constituent needed by one partial phrase matches the category of a completed phrase and the DAG associated with the completed phrase can be unified in appropriately.*

Scanner step: If $i \neq 0$ and $w_i = a$, then for all items $[h, i-1, X_0 \rightarrow \alpha.a\beta, D]$ add the item $[h, i, X_0 \rightarrow \alpha a.\beta, D]$.

Informally, this involves *allowing lexical items to be inserted into partial phrases.*

Notice that the Predictor Step in particular assumes the availability of the *cat* feature for top-down prediction. Consequently, this algorithm applies only to PATR-II with a context-free base.

5.4.3 Removing the Context-Free Base: An Inadequate Extension

A first attempt at extending the algorithm to make use of more than just a single atomic-valued *cat* feature (or less if no such feature is mandated) is to change the Predictor Step so that instead of checking the predicted rule for a left-hand side that matches its *cat* feature with the predicting subphrase, we require that the whole left-hand-side sub-DAG unifies with the subphrase being predicted from. Formally, we have

[4] Note that this unification will fail if $D(\langle X_0\ cat \rangle) \neq E(\langle X_j\ cat \rangle)$ and no edge will be added, i.e., if the subphrase is not of the appropriate category for insertion into the phrase being built.

[5] One edge subsumes another edge if and only if the first three elements of the edges are identical and the fourth element of the first edge subsumes that of the second edge.

Predictor step: For each item ending at i of the form $[h, i, X_0 \rightarrow \alpha.X_j\beta, D]$ and each rule of the form $\langle X_0 \rightarrow \gamma, E\rangle$, add an edge of the form $[i, i, X_0 \rightarrow .\gamma, E \sqcup [X_0 : D(X_j)]]$ if the unification succeeds and this edge is not subsumed by another edge.

This step *predicts top-down all rules whose left-hand side matches the* DAG *of some constituent being looked for.*

Completer step: As before.

Scanner step: As before.

However, this extension does not preserve termination. Consider a "counting" grammar that records in the DAG the number of terminals in the string.[6]

$S \rightarrow T$:
  $\langle Sf\rangle = a.$
$T_1 \rightarrow T_2\ A$:
  $\langle T_1 f\rangle = \langle T_2 f f\rangle.$
$S \rightarrow A.$
$A \rightarrow a.$

Initially, the $S - T$ rule will yield the edge

$$[0, 0, X_0 - .X_1, \left[\begin{array}{ll} X_0: & \left[\begin{array}{ll} cat: & S \end{array}\right] \\ X_1: & \left[\begin{array}{ll} cat: & T \\ f: & a \end{array}\right] \end{array}\right]]$$

which in turn causes the predictor step to give

$$[0, 0, X_0 \rightarrow .X_1X_2, \left[\begin{array}{lll} X_0: & \left[\begin{array}{ll} cat: & T \\ f: & \boxed{1}_a \end{array}\right] & \\ X_1: & \left[\begin{array}{lll} cat: & T & \\ f: & \left[\begin{array}{ll} f: & \boxed{1} \end{array}\right] & \end{array}\right] & \\ X_2: & \left[\begin{array}{ll} cat: & A \end{array}\right] & \end{array}\right]]$$

yielding in turn

[6]Similar problems occur in natural language grammars when keeping *lists* of, say, subcategorized constituents or gaps to be found.

$$[0,0,X_0 \rightarrow .X_1X_2, \left[\begin{array}{ll} X_0: & \left[\begin{array}{ll} cat: & T \\ f: & \boxed{1}\left[\ f: \ a\ \right] \end{array}\right] \\ X_1: & \left[\begin{array}{ll} cat: & T \\ f: & \left[\ f: \ \left[\ f: \ \boxed{1}\ \right]\ \right] \end{array}\right] \\ X_2: & \left[\ cat: \ A\ \right] \end{array}\right]]$$

and so forth, ad infinitum.

5.4.4 Removing the Context-Free Base: An Adequate Extension

What is needed is a way of "forgetting" some of the structure we are using for top-down prediction. But this is just what restriction gives us, since a restricted DAG always subsumes the original, i.e., it has strictly less information. Taking advantage of this property, we can change the Predictor step to restrict the top-down information before unifying it into the rule's DAG.

Predictor step: For each item ending at i of the form $[h,i,X_0 \rightarrow \alpha.X_j\beta, D]$ and each rule of the form $\langle X_0 \rightarrow \gamma, E\rangle$, add an edge of the form $[i,i,X_0 \rightarrow .\gamma, E \sqcup (D(X_j)\lceil\Phi)]$ if the unification succeeds and this edge is not subsumed by another edge.

This step *predicts top-down all rules whose left-hand side matches the* restricted DAG *of some constituent being looked for.*

Completer step: As before.

Scanner step: As before.

This algorithm on the previous grammar, using a restrictor that allows through only the *cat* feature of a DAG, operates as before, but predicts the first time around the more general edge:

$$[0,0,X_0 \rightarrow .X_1X_2, \left[\begin{array}{ll} X_0: & \left[\begin{array}{ll} cat: & T \\ f: & \boxed{1}[] \end{array}\right] \\ X_1: & \left[\begin{array}{ll} cat: & T \\ f: & \left[\ f: \ \boxed{1}\ \right] \end{array}\right] \\ X_2: & \left[\ cat: \ A\ \right] \end{array}\right]] \quad .$$

Another round of prediction yields *this same edge*, so the process terminates immediately. Because the predicted edge is more general than (i.e., subsumes) all the infinite number of edges it replaced that were predicted under the nonterminating extension, it preserves *completeness.* On the other hand, because the predicted edge is not more general than the

rule itself, it permits no constituents that violate the constraints of the rule; therefore, it preserves *correctness*. Finally, because restriction has a finite range, the predictor step can only occur a finite number of times before building an edge identical to one already built; therefore, it preserves *termination*.

5.5 Applications

5.5.1 Some Examples of the Use of the Algorithm

The algorithm just described has been implemented and incorporated into the PATR-II Experimental System, a grammar development and testing environment for PATR-II grammars written in Zetalisp for the Symbolics 3600.

The following table gives some data suggestive of the effect of the restrictor on parsing efficiency. It shows the total number of active and passive edges added to the chart for five sentences of up to eleven words using four different restrictors. The first allowed only category information to be used in prediction, thus generating the same behavior as the unextended Earley's algorithm. The second added subcategorization information in addition to the category. The third added filler-gap dependency information as well so that the gap proliferation problem was removed. The final restrictor added verb form information. The last column shows the percentage of edges that were eliminated by using this final restrictor.

	Prediction				%
Sentence	*cat*	+ *subcat*	+ *gap*	+ *form*	elim.
1	33	33	20	16	52
2	85	50	29	21	75
3	219	124	72	45	79
4	319	319	98	71	78
5	812	516	157	100	88

Several facts should be kept in mind about the data above. First, for sentences with no Wh-movement or relative clauses, no gaps were ever predicted. In other words, the top-down filtering is in some sense maximal with respect to gap hypothesis. Second, the subcategorization information used in top-down filtering removed all hypotheses of constituents except for those directly subcategorized for. Finally, the grammar used contained constructs that would cause nontermination in the unrestricted extension of Earley's algorithm.

5.5.2 Other Applications of Restriction

This technique of restriction of complex-feature structures into a finite set of equivalence classes can be used for a variety of purposes.

First, parsing algorithms such as the above can be modified for use by grammar formalisms other than PATR-II. In particular, definite-clause grammars are amenable to this

technique, and it can be used to extend the Earley deduction of Pereira and Warren [21]. Pereira has used a similar technique to improve the efficiency of the BUP (bottom-up left-corner) parser [7] for DCG. LFG and GPSG parsers can make use of the top-down filtering device as well. FUG parsers might be built that do not require a context-free backbone.

Second, restriction can be used to enhance other parsing algorithms. For example, the ancillary function to compute LR closure—which, like the Earley algorithm, either does not use feature information, or fails to terminate—can be modified in the same way as the Earley predictor step to terminate while still using significant feature information. LR parsing techniques can thereby be used for efficient parsing of complex-feature-based formalisms. More speculatively, schemes for scheduling LR parsers to yield parses in preference order might be modified for complex-feature-based formalisms, and even tuned by means of the restrictor.

Finally, restriction can be used in areas of parsing other than top-down prediction and filtering. For instance, in many parsing schemes, edges are indexed by a category symbol for efficient retrieval. In the case of Earley's algorithm, active edges can be indexed by the category of the constituent following the dot in the dotted rule. However, this again forces the primacy and atomicity of major category information. Once again, restriction can be used to solve the problem. Indexing by the restriction of the DAG associated with the need permits efficient retrieval that can be tuned to the particular grammar, yet does not affect the completeness or correctness of the algorithm. The indexing can be done by discrimination nets, or specialized hashing functions akin to the partial-match retrieval techniques designed for use in Prolog implementations [16].

5.6 Conclusion

We have presented a general technique of restriction with many applications in the area of manipulating complex-feature-based grammar formalisms. As a particular example, we presented a complete, correct, terminating extension of Earley's algorithm that uses restriction to perform top-down filtering. Our implementation demonstrates the drastic elimination of chart edges that can be achieved by this technique. Finally, we described further uses for the technique—including parsing other grammar formalisms, including definite-clause grammars; extending other parsing algorithms, including LR methods and syntactic preference modeling algorithms; and efficient indexing.

We feel that the restriction technique has great potential to make increasingly powerful grammar formalisms computationally feasible.

Acknowledgments

The author is indebted to Fernando Pereira and Ray Perrault for their comments on earlier drafts of this work.

References

[1] Ades, A.E. and M.J. Steedman. On the order of words. *Linguistics and Philosophy 4*, 4 (1982), 517–558.

[2] Ford, M., J. Bresnan, and R. Kaplan. A competence-based theory of syntactic closure. In J. Bresnan (ed.). *The Mental Representation of Grammatical Relations.* MIT Press, Cambridge, Massachusetts, 1982.

[3] Gawron, J.M., J. King, J. Lamping, E. Loebner, E.A. Paulson, G.K. Pullum, I.A. Sag, and T. Wasow. Processing Engglish with a generalized phrase structure grammar. *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics,* University of Toronto, Toronto, Ontario, Canada, June 1982, 74–81.

[4] Gazdar, G., E. Klein, G.K. Pullum, and I.A. Sag. *Generalized Phrase Structure Grammar.* Blackwell Publishing, Oxford, England, and Harvard University Press, Cambridge, Massachusetts, 1985.

[5] Kaplan, R. and J. Bresnan. Lexical-functional grammar: a formal system for grammatic representation. In J. Bresnan (ed.). *The Mental Representation of Grammatical Relations.* MIT Press, Cambridge, Massachusetts, 1983.

[6] Kay, M. An algorithm for compiling parsing tables from a grammar. Xerox Palo Alto Research Center, Palo Alto, California, 1980.

[7] Matsumoto, Y., H. Tanaka, H. Hirakawa, H. Miyoshi, and H. Yasukawa. BUP: a bottom-up parser embedded in Prolog. *New Generation Computing 1* (1983), 145–158.

[8] Montague, R. The proper treatment of quantification in ordinary English. In R.H. Thomason (ed.). *Formal Philosophy.* Yale University Press, New Haven, Connecticut, 1974, 188–221.

[9] Pereira, F.C.N. Logic for natural language analysis. Technical Note 275, Artificial Intelligence Center, SRI International, Menlo Park, California, 1983.

[10] Pereira, F.C.N. and S.M. Shieber. The semantics of grammar formalisms seen as computer languages. *Proceedings of the Tenth International Conference on Computational Linguistics,* Stanford University, Stanford, California, July 1984.

[11] Pereira, F.C.N. and D.H.D. Warren. *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics,* Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1983.

[12] Shieber, S.M. The design of a computer language for linguistic interpretation. *Proceedings of the Tenth International Conference on Computational Linguistics,* Stanford University, Stanford, California, July 1984.

[13] Shieber, S.M. Criteria for designing computer facilities for linguistic analysis. To appear in *Linguistics.*

[14] Shieber, S.M. Sentence disambiguation by a shift-reduce parsing technique. *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics,* Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1983, 113–118.

[15] Shieber, S.M., H. Uszkoreit, F.C.N. Pereira, J.J. Robinson, and M. Tyson. The formalism and implementation of PATR-II. In *Research on Interactive Acquisition and Use of Knowledge.* Artificial Intelligence Center, SRI International, Menlo Park, California, 1983.

[16] Wise, M.J. and D.M.W. Powers. Indexing Prolog clauses via superimposed code words and field encoded words. *Proceedings of the 1984 International Symposium on Logic Programming,* Atlantic City, New Jersey, February 1984, 203–210.

Chapter 6

A Morphological Recognizer With Syntactic And Phonological Rules

This chapter was written by John Bear.

6.1 Introduction

In many natural language processing systems currently in use, the morphological phenomena are handled by programs which do not interpret any sort of rules, but rather contain references to specific morphemes, graphemes, and grammatical categories. Recently Kaplan, Kay, Koskenniemi, and Karttunen have shown how to construct morphological analyzers in which the descriptions of the orthographic and syntactic phenomena are separable from the code. This chapter describes a system that builds on their work in the area of phonology/orthography and also has a well-defined syntactic component which applies to the area of computational morphology for the first time some of the tools that have been used in syntactic analysis for quite a while.

This chapter has two main parts. The first deals with the orthographic aspects of morphological analysis, the second with its syntactic aspects. The orthographic phenomena constitute a blend of phonology and orthography. The orthographic rules given in this chapter closely resemble phonological rules, both in form and function, but because their purpose is the description of orthographic facts, the words *orthography* and *orthographic* will be used in preference to *phonology* and *phonological.*

The overall goal of the work described herein is the development of a flexible, usable morphological analyzer in which the rules for both syntax and spelling are (1) separate from the code, and (2) descriptively powerful enough to handle the phenomena encountered when working with texts of written language.

6.2 Orthography

The researchers mentioned above use finite-state transducers for stipulating correspondences between surface segments, and underlying segments. In contrast, the system described in this chapter does not use finite state machines. Instead, orthographic rules are interpreted directly, as constraints on pairings of surface strings with lexical strings.

The rule notation employed, including conventions for expressing abbreviations, is based on that described in Koskenniemi [1983,1984]. The rules actually used in this system are based on the account of English in Karttunen and Wittenburg [1983].

6.2.1 Rules

What follows is an inductive introduction to the types of rules needed. Some pertinent data will be presented, then some potential rules for handling these data. We shall also discuss the reasons for needing a weaker form of rule and indicate what it might look like.

Let us first consider some data regarding English /s/ morphemes:

ALWAYS -ES
box+s ⟷ boxes
class+s ⟷ classes
fizz+s ⟷ fizzes
spy+s ⟷ spies
ash+s ⟷ ashes
church+s ⟷ churches
ALWAYS -S
slam+s ⟷ slams
hit+s ⟷ hits
tip+s ⟷ tips
...
SOMETIMES -ES,
SOMETIMES -S
piano+s ⟷ pianos
solo+s ⟷ solos
do+s ⟷ does
potato+s ⟷ potatoes
banjo+s ⟷ banjoes or banjos
cargo+s ⟷ cargoes or cargos

Below are presented two possible orthographic rules for describing the foregoing data:

R1) + ⟶ e {x | z | y/i | s (h) | c h} _ s
R2) + ⟶ e {x | z | y/i | s (h) | c h | o} _ s

The first of these rules will be shown to be too weak; the second, in contrast, will be shown to be too strong. This fact will serve as an argument for introducing a second kind of rule.

Before describing how the rules should be read, it is necessary to define two technical terms. In phonology, one speaks of underlying segments and surface segments; in orthography, characters making up the words in the lexicon contrast with characters in word forms that occur in texts. The term *lexical character* will be used here to refer to a character in a word or morpheme in the lexicon, i.e., the analog of a phonological underlying segment. The term *surface character* will be used to mean a character in a word that could appear in text. For example, [l o v e + e d] is a string of lexical characters, while [l o v e d] is a string of surface characters.

We may now describe how the rules should be read. The first rule should be read roughly as, "a morpheme boundary [+] at the lexical level corresponds to an [e] at the surface level whenever it is between an [x] and an [s], or between a [z] and an [s], or between a lexical [y] corresponding to a surface [i] and an [s], or between an [s h] and an [s] or between a [c h] and an [s]." This means, for instance, that the string of lexical characters [c h u r c h + s] corresponds to the string of surface characters [c h u r c h e s] (forgetting for the moment about the possibility that other rules might also obtain). The second rule is identical to the first except for an added [o] in the left context.

When we say [+] corresponds to [e] between an [x] and an [s], we mean between a lexical [x] corresponding to a surface [x] and a lexical [s] corresponding to a surface [s]. If we wanted to say that it does not matter what the lexical [x] corresponds to on the surface, we would use [x/=] instead of just [x].

The rules given above get the facts right for the words that do not end in [o]. For those that do, however, Rule 1 misses on [do+s] $\Longleftrightarrow$ [does], [potato+s] $\Longleftrightarrow$ [potatoes]; Rule 2 misses on [piano+s] $\Longleftrightarrow$ [pianos], [solo+s] $\Longleftrightarrow$ [solos]. Furthermore, neither rule allows for the possibility of more than one acceptable form, as in [banjo+s] $\Longleftrightarrow$ ([banjoes] or [banjos]), [cargo+s] $\Longleftrightarrow$ ([cargoes] or [cargos]).

The words ending in [o] can be divided into two classes: those that take an [es] in their plural and third-person singular forms, and those that just take an [s]. Most of the facts could be described correctly by adopting one of the two rules, e.g., the one stating that words ending in [o] take an [es] ending. In addition to adopting this rule, one would need to list all the words taking an [s] ending as being irregular. This approach has two problems. First, no matter which rule is chosen, a very large number of words would have to be listed in the lexicon; second, this approach does not account for the coexistence of two alternative forms for some words, e.g., [banjoes] or [banjos].

The data and arguments just given suggest the need for a second type of rule. It would stipulate that such and such a correspondence is *allowed* but *not required.* An example of such a rule is given below:

R3) +/e allowed in context o _ s.

Rule 3 says that a morpheme boundary may correspond to an [e] between an [o] and an [s]. It also has the effect of saying that if a morpheme boundary ever corresponds to an [e],

it must be in a context that is explicitly allowed by some rule.

If we now have the two rules R1 and R3,

R1) + ⟶ e / {x | z | y/i | s (h) | c h} _ s
R3) +/e allowed in context *o* _ s

we can generate all the correct forms for the data given. Furthermore, for the words that have two acceptable forms for plural or third person singular, we get both, just as we would like. The problem is that we generate both forms whether we want them or not. Clearly some sort of restriction on the rules, or "fine tuning," is in order; for the time being, however, the problem of deriving both forms is not so serious that it cannot be tolerated.

So far we have two kinds of rules, those stating that a correspondence always obtains in a certain environment, and those stating that a correspondence is allowed to obtain in some environment. The data below argue for one more type of rule, namely, a rule stipulating that a certain correspondence never obtains in a certain environment.

DATA FOR CONSONANT DOUBLING
DOUBLING:
bar+ed ⟷ barred
big+est ⟷ biggest
refer+ed ⟷ referred
NO DOUBLING:
question+ing ⟷ questioning
hear+ing ⟷ hearing
hack+ing ⟷ hacking
BOTH POSSIBILITIES:
travel+ed ⟷ (travelled or traveled) both are allowed

In English, final consonants are doubled if they, "follow a single [orthographic] vowel and the vowel is stressed." [from Karttunen and Wittenburg 1983]. So for instance, in [hear+ing], the final [r] is preceded by two vowels, so there is no doubling. In [hack+ing], the final [k] is not preceded by a vowel, so there is no doubling. In [question+ing], the last syllable is not stressed so again there is no doubling.

In Karttunen and Wittenburg [1983] there is a single rule listed to describe the data. However, the rule makes use of a diacritic (') for showing stress, and words in the lexicon must contain this diacritic in order for the rule to work. The same thing could be done in the system being described here, but it was deemed undesirable to allow words in the lexicon to contain diacritics encoding information such as stress. Instead, the following rules are used. Ultimately, the goal is to have some sort of general mechanism, perhaps negative rule features, for dealing with this sort of thing, but for now no such mechanism has been implemented.

RULES FOR CONSONANT DOUBLING

"Allowed-type" rules

'+'/b allowed in context vV b _ vV[1]
'+'/c allowed in context vV c _ vV
'+'/d allowed in context vV d _ vV
'+'/f allowed in context vV f _ vV
'+'/g allowed in context vV g _ vV
'+'/l allowed in context vV l _ vV
'+'/m allowed in context vV m _ vV
'+'/n allowed in context vV n _ vV
'+'/p allowed in context vV p _ vV
'+'/r allowed in context vV r _ vV
'+'/s allowed in context vV s _ vV
'+'/t allowed in context vV t _ vV
'+'/z allowed in context vV z _ vV

"Disallowed-type" rules

'+'/b disallowed in context vV vV b _ vV
'+'/c disallowed in context vV vV c _ vV
'+'/d disallowed in context vV vV d _ vV
'+'/f disallowed in context vV vV f _ vV
'+'/g disallowed in context vV vV g _ vV
'+'/l disallowed in context vV vV l _ vV
'+'/m disallowed in context vV vV m _ vV
'+'/n disallowed in context vV vV n _ vV
'+'/p disallowed in context vV vV p _ vV
'+'/r disallowed in context vV vV r _ vV
'+'/s disallowed in context vV vV s _ vV
'+'/t disallowed in context vV vV t _ vV
'+'/z disallowed in context vV vV z _ vV

The allowed-type rules in the top set are those that license consonant doubling. The disallowed-type rules in the second set constrain the doubling so it does not occur in words like [eat+ing] $\Longleftrightarrow$ [eating] and [hear+ing] $\Longleftrightarrow$ [hearing]. The disallowed-type rules say that a morpheme boundary [+] may not ever correspond to a consonant when the [+] is followed by a vowel and preceded by that same consonant and then two more vowels.

The rules given above suffer from the same problem as the previous rules, namely, over generation. Although they produce all the right answers and allow multiple forms for words like [travel+er] $\Longleftrightarrow$ ([traveller] or [traveler]), which is certainly a positive result, they also allow multiple forms for words which do not allow them. For instance they generate both [referred] and [refered]. As mentioned earlier, this problem will be tolerated for the time being.

[1] In these rules, the symbol vV stands for any element of the following set of orthographic vowels: {a,e,i,o,u}.

6.2.2 Comparison with Koskenniemi's Rules

Koskenniemi [1983, 1984] describes three types of rules, as exemplified below:

R4) a > b $\Rightarrow$ c/d e/f _ g/h i/j
R5) a > b $\Leftarrow$ c/d e/f _ g/h i/j
R6) a > b $\Leftrightarrow$ c/d e/f _ g/h i/j.

Rule R4 says that if a lexical [a] corresponds to a surface [b], then it must be within the context given, i.e., it must be preceded by [c/d e/f] and followed by [g/h i/j]. This corresponds exactly to the rule given below:

R7) a/b allowed in context c/d e/f _ g/h i/j.

The rule introduced as R5 and repeated below says that if a lexical [a] occurs following [c/d e/f] and preceding [g/h i/j], then it must correspond to a surface [b]:

R5) a > b $\Leftarrow$ c/d e/f _ g/h i/j.

The corresponding rule in the formalism being proposed here would look approximately like this:

R10) a/sS disallowed in context c/d e/f _ g/h i/j,

where sS is some set of characters to which [a]
can correspond that does not include [b].

A comparison of each system's third type of rule involves compostion of rules and is the subject of the next section.

6.2.3 Rule Composition and Decomposition

In Koskenniemi's systems, rule composition is fairly straightforward. Samples of the three types of rules are repeated here:

R4) a > b $\Rightarrow$ c/d e/f _ g/h i/j
R5) a > b $\Leftarrow$ c/d e/f _ g/h i/j
R6) a > b $\Leftrightarrow$ c/d e/f _ g/h i/j

If a grammar contains the two rules, R4 and R5, they can be replaced by the single rule R6.

In contrast, the composition of rules in the system proposed here is slightly more complicated. We need the notion of a default correspondence. The default correspondence for any alphabetic character is itself. In other words, in the absence of any rules, an alphabetic

character will correspond to itself. There may also be characters that are not alphabetic, e.g., the [+] representing a morpheme boundary, currently the only non-alphabetic character in this system. Other conceivable non-alphabetic characters would be an accent mark for representing stress, or say, a hash mark for word boundaries. The default for these characters is that they correspond to 0 (zero). Zero is the name for the null character used in this system.

Now it is easy to say how rules are composed in this system. If a grammar contains both R11 and R12 below, then R13 may be substituted for them with the same effect:

R11) a/b allowed in context c/d e/f _ g/h i/j
R12) a/ *"a's default"* disallowed in context c/d e/f _ g/h i/j
R13) a ⟶ b / c/d e/f _ g/h i/j

In fact, when a file of rules is read into the system, occurrences of rules like R13 are internalized as if the grammar really contained a rule like R11 and another like R12.

6.2.4 Using the Rules

Again consider for an example the rule R1 repeated below.

R1) + ⟶ e / {x | z | y/i | s (h) | c h} _ s

When this rule is read in, it is expanded into a set of rules whose contexts do not contain disjunction or optionality. Rules R14 through R19 are the result of the expansion:

R14) '+' ⟶ e / x _ s
R15) '+' ⟶ e / z _ s
R16) '+' ⟶ e / y/i _ s
R17) '+' ⟶ e / s _ s
R18) '+' ⟶ e / s h _ s
R19) '+' ⟶ e / c h _ s.

R14 through R19 are in turn expanded automatically into R20 through R31 below:

R20) '+'/0 disallowed in context x _ s
R21) '+'/0 disallowed in context z _ s
R22) '+'/0 disallowed in context y/i _ s
R23) '+'/0 disallowed in context s _ s
R24) '+'/0 disallowed in context s h _ s
R25) '+'/0 disallowed in context c h _ s
R26) '+'/e allowed in context x _ s
R27) '+'/e allowed in context z _ s
R28) '+'/e allowed in context y/i _ s

R29) '+'/e allowed in context s _ s
R30) '+'/e allowed in context s h _ s
R31) '+'/e allowed in context c h _ s.

The disallowed-type rules given here stipulate that a morpheme boundary, lexical [+], may never be paired with a null surface character, [0], in the environments indicated. Another way to describe what disallowed-type rules do, in general, is to say that they expressly rule out certain sequences of pairs of letters. For example, R20

R20) +/0 disallowed in context x _ s

states that the sequence

```
...x + s...
   | | |
...x 0 s...
```

is never permitted to be a part of a mapping of a surface string to a lexical string.

The allowed-type rules behave slightly differently than their disallowed-type counterparts. A rule such as

R26) '+'/e allowed in context x _ s

says that lexical [+] is not normally allowed to correspond to surface [e]. It also affirms that lexical [+] may appear between an [x] and an [s]. Other rules starting with the same pair say, in effect, "here is another environment where this pair is acceptable." The way these rules are to be interpreted is that a rule's main correspondence, i.e., the character pair that corresponds to the underscore in the context, is forbidden except in contexts where it is expressly permitted by some rule.

Once the rules are broken into the more primitive allowed-type and disallowed-type rules, there are several ways in which one could try to match them against a string of surface characters in the recognition process. One way would be to wait until a pair of characters was encountered that was the main pair for a rule, and then look backwards to see if the left context of the rule matches the current analysis path. If it does, put the right context on hold to see whether it will ultimately be matched.

Another possiblility would be to continually keep track of the left contexts of rules that are matching the characters at hand, so that when the main character of a rule is encountered, the program already knows that the left context has been matched. The right context still needs to be put on hold and dealt with the same way as in the other scheme.

The second of the two strategies is the one actually employed in this system, though it may very well turn out that the first one is more efficient for the current grammar of English.

6.2.5 Possible Correspondences

The rules act as filters to weed out sequences of character pairs, but before a particular mapping can be weeded out, something needs to propose it as being possible. There is a list — called a list of possible correspondences, or sometimes, a list of feasible pairs — that tells which characters may correspond to which others. Using this list, the recognizer generates possible lexical forms to correspond to the input surface form. These can then be checked against the rules and against the lexicon. If the rules do not weed it out, and it is also in the lexicon, we have successfully recognized a morpheme.

6.3 Syntax

The goal of the work being described was an analyzer that would be easy to use. In the area of syntax, this entails two subgoals. First, it should be easy to specify which morphemes may combine with which, and second, when the recognition has been completed, the result should be something that can easily be used by a parser or some other program.

Karttunen [1983] and Karttunen and Wittenburg [1983] have some suggestions for what a proper syntactic component for a morphological analyzer might contain. They mention using context-free rules and some sort of feature-handling system as possible extensions of both their and Koskenniemi's systems. In short, it has been acknowledged that any such system really ought to have some of the tools that have been used in syntax proper.

The first course of action that was followed in building this analyzer was to implement a unification system for dags (directed acyclic graphs), and then to have the analyzer unify the dags of all the morphemes encountered in a single analysis. That scheme turned out to be too weak to be practical. The next step was to implement a PATR rule interpreter [Shieber, et al. 1983] so that selected paths of dags could be unified. Finally, when that turned out to be still less flexible than one would like, the capability of handling disjunction in the dags was added to the unification package, and the PATR rule interpreter [Karttunen 1984].

The rules look like PATR rules with the context free skeleton. The first two lines of a rule are just a comment, however, and are not used in doing the analysis. The recognizer starts with the dag [cat: empty]. The rule below states that the "empty" dag may be combined with the dag from a verb stem to produce a dag for a verb.

```
% verb ⟶ empty + verb_stem
%   1          2          3
<2 cat> = empty
<3 cat> = verb_stem
<3 type> = regular
<1 type> = <3 type>
<1 cat> = verb
<1 word> = <3 lex>
<1 form> = {inf
```

[tense: pres
pers: {1 2}] } .

The resulting dag will be ambiguous between an infinitive verb and a present tense verb that is in either the first or second person. (The braces in the rule are the indicators of disjunction.) The verb stem's value for the feature *lex* will be whatever spelling the stem has. This value will then be the value for the feature *word* in the new dag.

The analyzer applies these rules in a very simple way. It always carries along a dag representing the results found thus far. Initially this dag is [cat: empty]. When a morpheme is found, the analyzer tries to combine it, via a rule, with the dag it has been carrying along. If the rule succeeds, a new dag is produced and becomes the dag carried along by the analyzer. In this way the information about which morphemes have been found is propagated.

If an [ing] is encountered after a verb has been found, the following rule builds the new dag. It first makes sure that the verb is infinitive (form: inf) so that the suffix cannot be added onto the end of a past participle, for instance, and then makes the tense of the new dag be pres_part for present participle. The category of the new dag is *verb*, and the value for *word* is the same as it was in the original verb's dag. The form of the input verb is a disjunction of *inf* (infinitive) with [tense: pres, pers: {1 2}], so the unification succeeds.

```
%  verb ⟶ verb + ing
%    1        2     3
<2 cat> = verb
<3 lex> = ing
<2 form> = inf
<1 cat> = verb
<1 word> = <2 word>
<1 form> = [tense: pres_part] .
```

The system also has a rule for combining an infinitive verb with the nominalizing [er] morpheme, e.g., swim : swimmer. This rule, given below, also checks the form of the input verb to verify that it is infinitive. It makes the resulting dag have *category: noun, number: singular*, and so on.

```
%  noun ⟶ verb + er
%    1        2     3
<2 cat> = verb
<3 lex> = er
<2 form> = inf
<1 cat> = noun
<1 word> = <2 word>
<1 nbr> = sg
<1 pers> = 3 .
```

The noun thus formed behaves just the same as other nouns. In particular, a pluralizing [s] may be added, or a possessive ['s], or any other affix that can be appended to a noun.

There are other rules in the grammar for handling adjective endings, more verb endings, etc. Irregular forms are handled in a fairly reasonable way. The irregular nouns are listed in the lexicon with *form: irregular*. Other rules than the ones shown here refer to that feature; they prevent the addition of plural morphemes to words that are already plural. Irregular verbs are listed in the lexicon with an appropriate value for tense (not unifiable with inf) so that the test for infinitiveness will fail when it should. Irregular adjectives, e.g. good, better, best, are dealt with in an analogous manner.

6.4 Further Work

There are still some things that are not as straightforward as one would like. In particular, consider the following example. Let us suppose as a first approximation that one wanted to analyze the [un] prefix in English as combining with adjectives to yield new ones, e.g., unfair, unclear, unsafe. Suppose also that one wanted to be able to build past participles of transitive verbs (passives) into adjectives, so that they could combine with [un], as in uncovered, unbuilt, unseen.

What we would need, would be a rule to combine an "empty" with an [un] to make an [un] and then a rule to combine an [un] with a verb stem to form a thing1, and finally a rule to combine a thing1 with a past participle marker to form a negative adjective. More rules would be needed for the case where [un] combines with an adjective stem like [fair]. In addition, rules would be needed for irregular passives, etc.

In short, without a more sophisticated control strategy, the grammar would contain a fair amount of redundancy if one really attempted to handle English morphology in its entirety. However, on a more positive note, the rules do allow one to deal effectively and elegantly with a sufficient range of phenomena to make it quite acceptable as, for instance, an interface between a parser and its lexicon.

6.5 Conclusion

A morphological analyzer has been presented that is capable of interpreting both orthographic and syntactic rules. This represents a substantial improvement over the method of incorporating morphological facts directly into the code of an analyzer. The use of these rules leads to a powerful, flexible morphological analyzer.

Acknowledgments

The author is indebted to Lauri Karttunen and Fernando Pereira for all their help. Lauri supplied the initial English automata on which the orthographic grammar was based, while Fernando furnished some of the Prolog code. Both provided many helpful suggestions and

explanations as well. Thanks also go to Kimmo Koskenniemi for his comments on an earlier draft of this section.

References

[1] Karttunen, L. (1983) "Kimmo: A General Morphological Processor," in *Texas Linguistic Forum* #22, Dalrymple et al., eds., Linguistics Department, University of Texas, Austin, Texas.

[2] Karttunen, L. (1984) "Features and Values," in *COLING 84*.

[3] Karttunen, L. and K. Wittenburg (1983) "A Two-level Morphological Analysis Of English," in *Texas Linguistic Forum* #22, Dalrymple et al., eds., Linguistics Department, University of Texas, Austin, Texas.

[4] Kay, M. (1983) "When Meta-rules are not Meta-rules," in K. Sparcke-Jones, and Y. Wilkes, eds. *Automatic Natural Language Processing*, John Wiley and Sons, New York.

[5] Koskenniemi, K. (1983) "Two-level Model for Morphological Analysis," *IJCAI 83*, pp. 683-685.

[6] Koskenniemi, K. (198 , "A General Computational Model for Word-form Recognition and Production," *COLING 84*, pp. 178-181.

[7] Selkirk, E. (1982) *The Syntax of Words*, MIT Press.

[8] Shieber, S., H. Uszkoreit, F. Pereira, J. Robinson, and M Tyson (1983) "The Formalism and Implementation of PATR-II," in B. Grosz, and M. Stickel (1983) *Research on Interactive Acquisition and use of Knowledge*, SRI Final Report 1894, SRI International, Menlo Park, California.

99

Chapter 7

P-PATR: A Compiler for Unification-Based Grammars

This chapter was written by Susan Hirsh.

7.1 Introduction and Motivations

P-PATR is a compiler for unification-based grammars that is written in Quintus Prolog running on a Sun 2 workstation. P-PATR is based on the PATR-II[1] formalism [14] developed at SRI International. PATR is a simple, unification-based formalism capable of encoding a wide variety of grammars. As a result of this versatility, several parsing systems and development environments based on this formalism have been implemented [18,5]. P-PATR is one such system, designed in response to the slow parse times of most of the other PATR implementations.

Most of the currently running PATR systems operate by *interpreting* a PATR grammar. P-PATR differs from these systems by *compiling* the grammar into a Prolog definite-clause grammar (DCG) [8].

The compilation is done only once for a given grammar and the DCG produced as a result contains all the information in the original PATR grammar in a form readily conducive to parsing. The advantage of compilation is that less work needs to be done during parsing as some of the necessary computations have already been done in the compilation phase.

The use of Prolog as the target language of the compiler is advantageous for two reasons. Prolog, like PATR, uses unification as its method of operation. By compiling the PATR grammar into Prolog, P-PATR takes advantage of the efficient implementation of Prolog unification. Secondly, the performance of the resulting DCG can be improved further by compiling it with a Prolog compiler.

The compilation combined with the use of Prolog give P-PATR a speed advantage over the other currently implemented PATR systems.

[1] Henceforth referred to simply as PATR.

The rest of this discussion is divided into three parts. The first section discusses the basic algorithm used in compiling the PATR grammar into a Prolog DCG. The second part goes into excruciating detail describing the actual procedure followed during the compilation. Appendix C contains a user manual to the P-PATR system as well as a sample grammar and some selected Prolog code from the system.

7.2 Methods

What follows is a detailed explanation of the techniques used in compiling a PATR grammar into a Prolog DCG. First, an explanation of the general mechanisms used in compiling a PATR grammar into a DCG is given. This compilation scheme is then refined so that the DCG produced is equivalent to the original PATR grammar.

7.2.1 Feature Structures as Prolog Terms

In Prolog, unification operates on terms, and not on PATR feature structures. It is therefore necessary to model PATR feature structures as Prolog terms to take advantage of the Prolog unification mechanism.

Prolog terms differ from PATR feature structures in two major ways [14]. First, in a Prolog term a value is identified by its position, while PATR feature structures identify a value by associating them with an attribute. For example, the two Prolog terms:

```
head( agreement( number( plural ), person( third ) ) )
head( agreement( person( third ), number( plural ) ) )
```

do not unify. Because the order of the arguments is different, *number(plural)* is matched against *person(third)* and the unification fails. The second difference is that two Prolog terms only unify if they have the same number of arguments, whereas two PATR feature structures may unify even if they differ in the number of features. For example, the two terms:

```
(1) head( agreement( number( plural ), person( third ) ) )
(2) head( agreement( number( plural ) ) )
```

do not unify because the arities do not match. Thus, in representing a feature structure as a Prolog term, each structure must be given a fixed order and arity.

There are two methods generally used in modeling feature structures as Prolog terms. They will be referred to as *tailing* and *feature precompilation.*

- *Tailing*

  The first method of conversion of feature structures to Prolog terms involves the use of tail variables. Each feature structure is encoded as a Prolog term of the form[2]:

  ```
  [ feature1: value1, ... , featureN: valueN | T ]
  ```

[2] Using the Prolog list notation to represent a list with an uninstantiated tail variable [3].

where an uninstantiated tail variable is placed at the end of the list. Then, as this structure is unified with new structures, the features in the new structure are reordered corresponding to the features seen so far, and any new features are unified with the tail variable. For example, feature structures 1 and 2 are represented as the Prolog terms[3]:

```
[ head: [ agreement: [ number: plural,
                       person: third | T1 ] | T2 ] | T3 ]

[ head: [ agreement: [ number: plural | T4 ] | T5 ] | T6 ]
```

and then when unified, *person: third* unifies with the tail variable in the *agreement* list, producing the new Prolog term:

```
[ head: [ agreement: [ number: plural,
                       person: third | T1 ] | T2 ] | T3 ]
```

- *Feature precompilation*

The second conversion method involves a preliminary pass through the grammar to determine the arity and composition of all complex feature values. On the second pass, every attribute-value pair is placed in the correct position and order with respect to the other features. If a feature is missing from the structure, an uninstantiated variable is inserted in its place. For example, from feature structures 1 and 2 the following information is extracted:

head can be followed by the feature *agreement.*
agreement can be followed by the two features:
number and *person*, in that order.

These feature structures are converted into the Prolog terms[4]:

```
[ head: [ agreement: [ number: plural, person: third ] ] ]
[ head: [ agreement: [ number: plural, person: X ] ] ]
```

where the missing *person* value in feature structure 2 is represented by the uninstantiated variable X. The two Prolog terms now unify successfully to:

```
[ head: [ agreement: [ number: plural, person: third ] ] ]
```

[3]This is not quite accurate. Throughout this chapter feature structures are represented by labeling the values with the attribute they represent, but only the values of the attributes are actually present in the feature structures. The attributes are included just for readability.

[4]Variables are distinguished from atoms by an initial uppercase letter.

P-PATR uses the feature precompilation method described above in encoding the feature structures associated with the PATR grammar entries as Prolog terms. When the unification list of a rule is processed during compilation, this feature information is used in creating the feature structures. For example, given the information extracted from feature structures 1 and 2, the unification:

<X head agreement person> = <Y head agreement person>

produces the feature structures for X and Y:

[head: [agreement: [person: A, number: B]]]
[head: [agreement: [person: A, number: D]]]

where the values of the *person* attribute are unified and the indeterminate values for *number* are added to complete the *agreement* features.

7.2.2 Basic Compilation

The compilation produces a DCG that stands in a one-to-one relationship with the original PATR grammar.

- *Grammar rules*

  PATR grammar rules consist of a context-free phrase structure (CFPS) rule augmented with a list of unifications. For example:

  S → NP VP:
  <S head> = <VP head>
  <VP head agreement> = <NP head agreement>.

  The CFPS part of the rule is:

  S → NP VP

  and the unifications give the added information that the agreement features of the VP and the NP must be the same.

  DCGs are a natural extension of context-free grammars (CFG); a straightforward translation scheme is given in [7]. The constituents of a DCG rule may be complex symbols, consisting of a functor and a list of arguments. In the translation of a PATR rule to a DCG rule, the CFPS part of the rule provides the functors of the DCG rule, while the feature structure information from the unifications is encoded as the arguments to these functors. For example, the grammar rule just presented is equivalent to the DCG rule[5]:

[5] Reentrancy is represented by sharing variables.

```
s([ head: [ agreement: Y ] ]) --> np([ head: [ agreement: Y ] ]),
                                  vp([ head: [ agreement: Y ] ]).
```

- *Lexical Entries*

  PATR lexical entries consist of a word followed by a list of unifications. For example:

  ```
  Word Uther:
       <cat> = NP
       <head agreement person> = third
       <head agreement number> = singular.
  ```

  This entry defines the word "Uther"; the unifications encode the information that "Uther" is a third-person singular NP.

  In translating a PATR lexical entry into a DCG rule, the category of the word becomes the functor for the left-hand side (Lhs) of the rule; its argument list is derived from the list of unifications. The right-hand side (Rhs) of the rule consists of the word itself. For example, the above lexical entry is equivalent to the DCG rule:

  ```
  np([ head: [ agreement: [ person: third, number: singular ] ] ])-->
    [ uther ].
  ```

The example just presented shows a very simple correspondence between the PATR and the DCG formalisms. For reasons explained in the next sections, P-PATR actually uses a more complex compilation technique.

7.2.3 Left-Corner Parsing

The default parsing algorithm for DCGs supplied by Prolog is a top-down, left-to-right, backtrack algorithm. A well-known problem with top-down parsers is that left-recursive grammars can cause them to go into an infinite loop [1]. Because PATR rules are allowed to be left recursive, a compilation technique must be applied that enables the Prolog DCG to handle such rules.

P-PATR compiles a PATR grammar into a DCG that uses a bottom-up parsing algorithm. Bottom-up parsers have no problem with left recursion [1]. The particular parsing technique used is called left-corner parsing [11].

The left corner (LC) of a CFG rule is the first symbol of the right-hand side of the rule. For example, the LC of the rule:

$$S \rightarrow NP\ VP$$

is the nonterminal NP. In LC parsing, each rule is identified through its LC. The first word in the sentence functions as the initial LC key. The rules whose LC match the key

are extracted. The next word in the sentence becomes the new LC key for satisfying the remainder of the right-hand side of these rules. If the right-hand side of the rule is completely satisfied, the left-hand side of the rule is substituted for the LC key and the process is iterated. For example, given the CFG rules:

(3) S → NP VP
(4) VP → V
(5) NP→ N
(6) V → sleeps
(7) N → Bill

the sentence "Bill sleeps" is parsed as follows:

LC key = "Bill"
matches the LC of rule 7,
rule 7 is satisfied

LC key = N
matches the LC of rule 5,
rule 5 is satisfied

LC key = NP
matches the LC of rule 3,
leaving the VP of rule 3 to be satisfied

LC key = "sleeps"
matches the LC of rule 6,
rule 6 is satisfied

LC key = V
matches the LC of rule 4,
rule 4 is satisfied

rule 3 is satisfied,
no more input,
parse successful

This parsing algorithm avoids the problem of left-recursive rules[6].

The DCG produced by P-PATR is based on the implementation of the LC algorithm in [6]. Each PATR grammar rule of the type:

$$Lhs \rightarrow Rhs_1 \ldots Rhs_n$$

[6]Epsilon rules still pose a problem, but they are taken care of separately (Section 7.2.4).

is converted into a DCG rule of the form:

```
lc( Rhs1, Root ) -->
  down( Rhs2 ), ... , down( RhsN ),
  lc( Lhs, Root ).
```

where Lhs and Rhs1 ... RhsN are the feature structure information from the unifications list of the rule and Root is the feature structure of the constituent currently being parsed. In the limit case, Root and Lhs are the same: everything is its own LC.

```
lc( Root, Root ) --> [].
```

For example, consider a CFG for noun phrases consisting of a rule:

NP → Det N

and two lexical items: the:Det and girl:N. The corresponding DCG rule produced by P-PATR is:

```
lc( det, Root ) -->
  down( n ),
  lc( np, Root ).
```

To understand how this rule is used by the Prolog parser, we first need to define the predicates **down** and **leaf**.

```
down( Cat ) -->
      leaf( Child )
      lc( Child, Cat ).

leaf( Child ) -->
    [ Word ],
    {lex( Word, Child )}.
```

The two words in the grammar are defined by the following Prolog clauses:

```
lex( the, det ).
lex( girl, n ).
```

Let us now see how the string "the girl" is parsed as an NP using this DCG version of the original CFG.

The parse is initiated with the call:

down(np).

This results in the call:

leaf(Child).

which consumes the word "the" and binds the variable Child to the word's category *det*. The next step is to evaluate the call:

lc(det, np).

by finding a match for this clause among the LC rules and satisfying the right-hand side of the LC rule. In this case, we need to satisfy the calls:

down(n).
lc(np, np).

The first clause triggers another call to:

leaf(Child).

which now consumes the word "girl" and binds Child to *n*, and the call:

lc(n, n).

which is immediately satisfied because it is an instance of the rule:

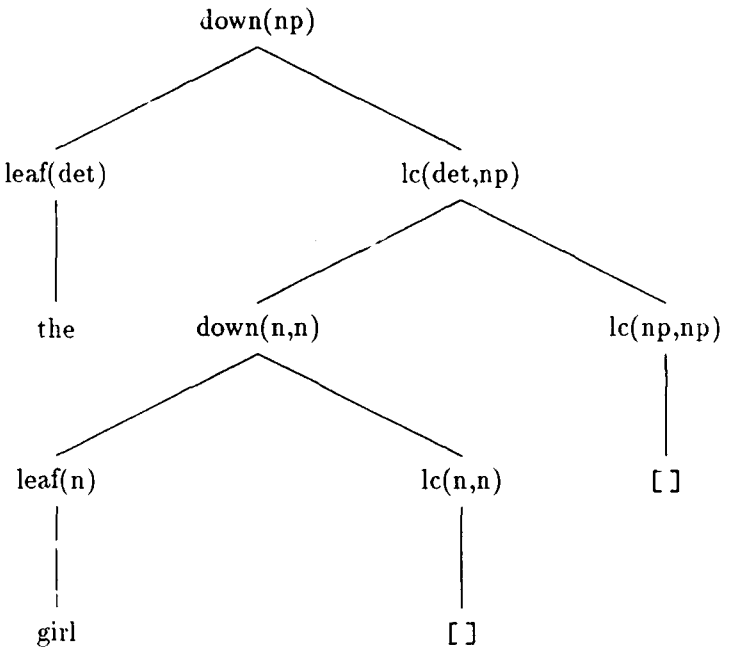
lc(Root, Root) --> [].

leaving the call:

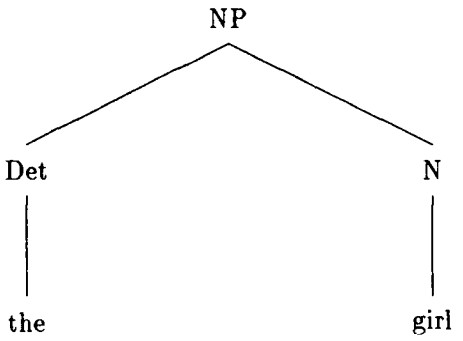
lc(np, np).

to be satisfied in the same way.

The flow of the computation can be pictured as the tree:



This obviously differs from the usual parse tree:



because of the way the LC algorithm uses the rules. The standard phrase structure tree can easily be produced as a side effect of the parse, if desired.

The above discussion is an oversimplification. In actuality, the values of the variables Root, Cat and Child are feature structures rather than atomic category symbols. For example, the grammar rule presented above becomes the new DCG rule[7]:

[7]After feature information corresponding to the categories of the nonterminals is added to the feature structures (Section 7.3.1).

```
lc([ cat: np, head: [ agreement: Y ] ], Root ) -->
  down([ cat: vp, head: [ agreement: Y ] ]),
  lc([ cat: s, head: [ agreement: Y ] ]), Root ).
```

and the lexical entry for "Uther" presented becomes the Prolog clause:

```
lex( uther,[ cat: np, head: [ agreement: [ person: third, number: singular ] ]).
```

A PATR grammar is compiled into a DCG of the form just presented. The compilation technique is revised slightly in the next section to allow for the epsilon rules that produce empty constituents.

7.2.4 Epsilon Rules

Epsilon rules in a CFG are of the form:

$$A \rightarrow \epsilon$$

This type of rule can pose a problem to the compilation technique described above. In LC parsing, a rule is keyed by its left corner. If the LC of a rule can be expanded to an empty string, the rule in effect has a second left corner.

For example, consider the rules:

(8) $A \rightarrow B\ A$
(9) $B \rightarrow \epsilon$

Because B can be expanded by rule 9 to an empty string, rule 8 has two left corners: B and A. For the compilation technique described above to work, each possible LC has to be recognized before a rule is compiled.

The problem is solved in two stages. First, all epsilon rules are extracted from the grammar and put into a separate list. Then all of the remaining rules are examined one by one. If the LC of a rule:

$$Lhs \rightarrow Rhs_1 \ldots Rhs_n$$

can be null, a new rule of the form:

$$Lhs \rightarrow Rhs_2 \ldots Rhs_n$$

is added to the grammar and subjected to the same test. For example, rule 8 above gives rise to the new rule:

A → A

on account of the possible expansion of B in rule 8 by rule 9.

The technique outlined above is easily extended to PATR grammars. In a PATR grammar an epsilon rule is of the form:

A → :
<Definition>.

In eliminating the epsilon rules, the unification information must be taken into account. For example, for the PATR grammar rules:

(10) A → B A:
<A feature1> = value1
<B feature2> = <A feature2>.

(11) B →:
<B feature2> = value2.

rule 10 gives rise to the new rule:

A → A:
<A feature1> = value1
<A feature2> = value2.

7.2.5 Lexical Organization

We now turn to *lexical templates* and *lexical rules*. Lexical templates are named feature structures and lexical rules are named transformations on feature structures. Both types of entries may include references to templates and rules in their definition. Because templates and rules may be referenced before they are defined, the compilation takes place in two stages.

- *Compilation: First Stage*

  In the first stage, each lexical entry of the PATR grammar is compiled into a temporary DCG rule of the form:

  word(Word, FeatureStructure):- ...

  The right-hand side of a temporary DCG rule typically contains references to the lexical templates and lexical rules that occur in the entry. These references cannot be evaluated until the first stage is completed. The references are of the form:

```
template( Name, In, Out )
        or
lex_rule( Name, In, Out )
```

where In is the input feature structure to a rule or template, and Out is the output feature structure from the rule or template.

For example, a lexical entry:

```
Word Uther:
    noun.
```

is compiled into the temporary DCG rule:

```
word( uther, FeatureStructure ):-
    template( noun, In, FeatureStructure ).
```

- *Compilation: Second Stage*

  Once the first stage is completed, the definitions of the lexical rules and lexical templates reside in the Prolog database (Section 7.3.2). The temporary rules produced in the first stage of compilation could be used by the parser, but this would be inefficient because the lexical templates and rules would be executed each time they are referenced.

  At this point, each lexical entry is executed once by Prolog, evaluating the actions of the rules and templates, and the new feature structure produced is used in converting the entry to its final form.

  For example, the temporary DCG rule:

```
word( boy, FeatureStructure ):-
    template( noun, In, FeatureStructure ).
```

  produces the final DCG rule:

```
lex( boy, [ cat: n ] ) ).
```

  once it is executed.

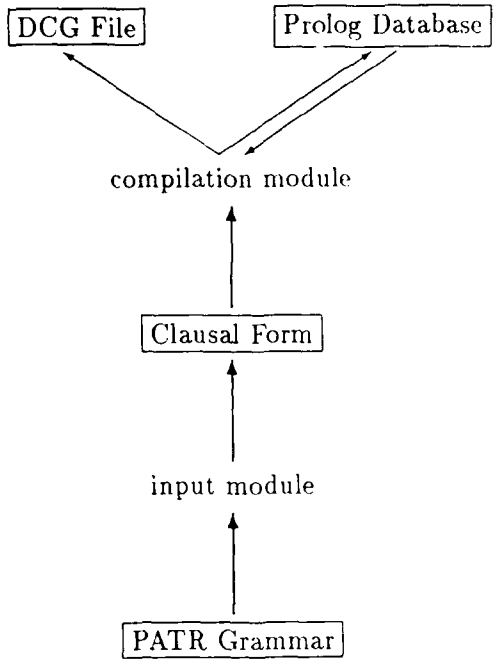


Figure 7.1: Flow diagram of P-PATR

7.3 The P-PATR System

This section gives a step-by-step account of the compilation technique used by P-PATR. An overview of the process is given in Figure 7.1. As shown in the diagram, compilation is accomplished in two phases: *grammar input* and *grammar compilation*. The grammar input phase produces an intermediate representation of the PATR grammar that is used in the compilation. During compilation, information is both written into a file reserved for the output DCG and asserted into the Prolog database. The information in the database is accessed as the compilation proceeds.

7.3.1 Grammar Input

This phase takes a set of text files containing a PATR grammar and converts it to a Prolog clausal form used by later phases. The grammar is input in two distinct steps: *tokenization* and *translation*.

Tokenization

Each entry in the PATR grammar is first tokenized and then translated to clausal form. There are six classes of tokens recognized by the P-PATR tokenizer: *identifiers*, *special characters*, *terminators*, *whitespace characters*, *comments* and *strings*. Each token type is briefly described below.

- *Identifiers*

  Identifiers are tokens that consist of any alphanumeric characters: a-z, A-Z, and 0-9; and any special inword characters: underbar (_), asterisk (*), apostrophe ('), questionmark (?), and backquote (`).

- *Special characters*

  Special characters are tokens consisting of a single character: colon (:), number sign (#), slash (/), arrow (→), square brackets ([,]), angle brackets (<, >), braces ({, }), parentheses ((,)), comma (,), equals sign (=), or dash(-). A sequence of tokens consisting of a dash (-) and a right angle-bracket (>) is treated as the single token: arrow (→).

- *Terminators*

  Terminators: period (.) and end of file, signal the end of a token stream. Terminators are considered a special case of special characters and are treated as the single tokens: period (.) and end_of_file.

- *Whitespace characters*

  Whitespace characters: space, newline, tab and formfeed, are ignored.

- *Comments*

  Comments, which begin when the single token semicolon (;) is encountered and continue to the end of the line, are ignored.

- *Strings*

  Strings are any list of characters enclosed in double quotes ("). Embedding of double quotes inside a string is done by using a sequence of two double quotes ("").

In all tokens, except for strings, no case distinction is made. All characters are converted to lowercase. Any characters that are not legal in a P-PATR token are ignored and a warning is given.

Translation

The stream of tokens produced by the tokenization process is now translated to clausal form. Each type of entry in the PATR grammar is translated into a form that will be most appropriate in further compilation (Section 7.3.2), as follows:

Control statements

The only type of control statement is the input statement. Input statements are of the form

Input <InputFile>.

When an input statement is encountered during translation, the current input file is temporarily replaced by the file specified in the statement. Once this new file is completely read in, the old input file is restored.

For example, the input

```
Input 'testgram'.
```

causes the current input stream to become the file TESTGRAM.

Parameters

P-PATR recognizes two grammar-dependent parameters: *start symbol* and *attribute order.* These parameters are set by statements that must appear in the grammar before any rules or lexical items are encountered. Other parameters[8] are ignored.

The parameter statements are processed as follows:

- *Start symbol*

  The start symbol is defined by a statement of the form

  ```
  Parameter: Start Symbol is <Symbol>.
  ```

  The start symbol for the grammar is recorded for use in further compilation as a clause of the form

  ```
  parameter( start( Symbol ) ).
  ```

- *Attribute order*

  Attribute order is specified as follows:

  ```
  Parameter: Attribute Order is <Attributes>.
  ```

  This is converted to the Prolog clause

  ```
  parameter( attributes( List ) ).
  ```

  where List corresponds to a list of all attributes in the order specified.

  For example, the input

  ```
  Parameter: Attribute Order is cat head.
  ```

  produces the clause

  ```
  parameter( attributes( [ cat, head ] ) ).
  ```

[8]There are several other parameters that can be specified in a PATR grammar, but their information is not used by this implementation.

Grammar rules

The format for PATR grammar rules is

Rule { <Description> }
<Lhs> → <Rhs>:
<Definition>.

In translating the rule to clausal form all nonterminals are replaced by variables, which are used during compilation. Grammar rules are translated into a clause of the form

rule(Lhs, Rhs, Def).

where:
Lhs is a variable associated with the left-hand side of a rule.
Rhs is a list of variables associated with the right-hand side of the rule.
Def is a list of specifications defining the rule.

In the original PATR grammar the category information of a nonterminal can be omitted from the list of unifications because it is added automatically during grammar translation. For example, the grammar rule

Rule { sentence formation }
S → NP VP:
< S head > = < VP head>
< S head form > = finite
< VP subcat first > = < NP >
< VP subcat rest > = end.

produces the clause

```
rule( S, [ NP, VP ], [ [ S, cat ] = s,
                       [ NP, cat ] = np,
                       [ VP, cat ] = vp,
                       [ S, head ] = [ VP, head ],
                       [ S, head, form ] = finite,
                       [ VP, subcat, first ] = [ NP ],
                       [ VP, subcat, rest ] = end ] ).
```

where the unification information

[S, cat] = s
[NP, cat] = np
[VP, cat] = vp

is added to the list of unifications. The only exception is the nonterminal X (with or without a subscript). If this appears in a grammar rule no category information is added, allowing expressions of any category to appear in this position.

P-PATR follows the Z-PATR [18] convention for distinguishing between constituents that have the same category. This is accomplished by means of numeric tags. For example, if two constituents in the same rule are referred to as VP_1 and VP_2, they are both of category VP.

Lexical items

Each type of lexical item in a PATR grammar is translated accordingly:

- *Lexical entries*

  The format for lexical entries is

  Word <Word>:
  <Definition>.

  In translating a lexical entry to clausal form, the information from the original PATR entry is left unchanged. Thus, lexical entries are translated to clauses of the form

  lex(Word, Def).

  where Word is a word being defined, and Def is a list of specifications defining the word.

  The system augments each lexical entry with two new features: *lex* and *sense*. It is assumed that the lexical entry does not already contain this information, otherwise it will be duplicated. The *lex* value for a lexical entry is the word itself. The *sense* value is the word concatenated with a number that specifies how many previous definitions of this word have occurred in this grammar. For example, given the entry

  Word Uther:
  < cat > = NP
  < head agreement gender > = masculine
  < head agreement person > = third
  < head agreement number > = singular
  < head trans > = Uther.

  P-PATR produces the clause

```
lex( uther, [[ lex ] = uther,
             [ sense ] = uther,
             [ cat ] = np,
             [ head, agreement, gender ] = masculine,
             [ head, agreement, person ] = third,
             [ head, agreement, number ] = singular,
             [ head, trans ] = uther ] ).
```

If there already exists one previous definition for the word "Uther", the *sense* for the second definition is *uther2*.

- *Lexical templates*

  Lexical templates are of the form

  ```
  Let <Template> be
      <Definition>.
  ```

  In translating a lexical template to clausal form, the information from the original PATR lexical template is left unchanged. Thus, lexical templates are translated to clauses of the form

  ```
  template( Name, Def ).
  ```

  where Name is the name of a lexical template being defined, and Def is a list of specifications defining the template.

  For example, the template

  ```
  Let verb be
      < cat > = V.
  ```

  produces the clause

  ```
  template( verb, [ [ cat ] = v ] ).
  ```

- *Lexical rules*

  Lexical rules have the form

  ```
  Define <Rule> as
         <Definition>.
  ```

  In the clausal form encoding of the lexical rule, the *in* and *out* attributes are replaced by variables, which are used during compilation. Thus, lexical rules are translated to clauses of the form

lex_rule(Name, In, Out, Def)

where:
Name is the name of a lexical rule being defined.
In is a variable associated the with the input to the rule.
Out is a variable associated with the output of the rule.
Def is a list of specifications defining the rule.

For example, the rule

Define agentlesspassive as:
< out cat > = < in cat >
< out subcat > = < in subcat rest >
< out head agreement > = < in head agreement >
< out head aux > = < in head aux >
< out head trans > = < in head trans >
< out head form > = passiveparticiple.

produces the clause

```
lex_rule( agentlesspassive, In, Out,
          [[ Out, cat ] = [ In, cat ],
           [ Out, subcat ] = [ In, subcat, rest ],
           [ Out, head, agreement ] = [ In, head, agreement ],
           [ Out, head, aux ] = [ In, head, aux ],
           [ Out, head, trans ] = [ In, head, trans ],
           [ Out, head, form ] = passiveparticiple ] ).
```

7.3.2 Grammar Compilation

This phase takes a text file containing a PATR grammar in clausal form and compiles it into a Prolog DCG. Grammar compilation is accomplished in five distinct phases: *parameter processing*, *attribute position generation*, *epsilon precompilation*, *compilation* and *lexical compilation: second stage.*

Parameter processing

This phase processes the parameter statements specified in the PATR grammar. Parameter statements must occur first in the grammar, to insure their use in the entire compilation.

The two types of parameter statements are treated as follows:

- *Start symbol*

  A statement of the form

  ```
  parameter( start( Symbol ) ).
  ```

  is asserted and written into the DCG file as

  ```
  start( Symbol ).
  ```

- *Attribute order*

  The attribute order is initially represented in clausal form as

  ```
  parameter( attributes( List ) ).
  ```

  where List is a list of attributes with a specified order.

  For each attribute in the list, a clause is asserted into the Prolog database specifying the order of that attribute. This information is used in maintaining the specified order during the output of the feature structures.

  This information is asserted as

  ```
  print_order( Attribute, Position ).
  ```

  where Attribute is an attribute from the list of attributes, and Position is the position of that attribute in the list of attributes.

  For example, given the parameter statement

  ```
  parameter( attributes( [ cat, head ] ).
  ```

  the clauses

  ```
  print_order( cat, 1 ).
  print_order( head, 2 ).
  ```

  are asserted.

Attribute position generation

In PATR, features are pairs of attributes and values. The value of an attribute can be of three types: indeterminate, atomic and complex. A complex value is a set of attribute-value pairs. In the following only the complex values contribute information about the attributes, therefore the other types of values are not discussed.

This phase computes the arity of each complex attribute value and places the features in a fixed order. The information is used in the conversion of PATR feature structures to Prolog terms (Section 7.2.1).

For each attribute in a PATR grammar, a list is compiled that consists of all the attributes that can follow the given attribute in a path specification. For example, given the lexical template:

```
template( singular, [ [ head, agreement, number ] = singular ] ) .
```

the information recorded for the attribute *head* is that it can be followed by *agreement* in a path specification. The information that the attribute *agreement* can be followed by *number* is also recorded.

Once all information on the attributes is compiled, this information is translated into clausal form and is asserted and written into the DCG file as

```
feature_order( Attribute, Features, Variables ).
```

where:
Attribute is the attribute currently being described.
Features is a list of pairs consisting of an attribute and a unique
variable representing the value of that attribute.
Variables is a list of the variables in Features.

For example, from the template above the following clauses are generated and asserted:

```
feature_order( main, [ head:X ], [ X ] ).
feature_order( head, [ agreement:Y ], [ Y ] ).
feature_order( agreement, [ number:Z ], [ Z ] ).
```

A dummy attribute *main* is created to notate those features that can occur as the first feature in a path specification.

The list Features is used during the output of the feature structures, therefore the order of the attributes must reflect the order specified in the parameter statement. Thus, the list is reordered to reflect the specified order. Any attributes whose order is not determined are just added to the end of the list of features.

Epsilon precompilation

This pass through the PATR grammar precompiles epsilon rules.

Epsilon rules are represented in clausal form as

```
rule( Lhs, [], Def ).
```

where the grammar rule has no right-hand side. All other grammar entries are ignored during this pass.

An epsilon rule is compiled into a DCG rule by applying the unifications equations attached to the rule, producing a feature structure associated with the rule (Section 7.2.1). The compiled epsilon rule is then asserted and written into the DCG file as

```
null( Lhs ).
```

where Lhs is the feature structure associated with the rule.

For example, the epsilon rule

```
rule( Det, [], [[ Det, head, agreement, number ] = plural ] ).
```

is output as

```
null( [ cat: det, head: [ agreement: [ number: plural ] ] ] ).
```

Compilation

This pass through the PATR grammar uses the information produced in the previous phases to produce a DCG rule for each grammar entry. These DCG rules are written into the DCG file (grammar rules) or recorded in the database to be further processed during the second compilation stage (lexical items).

Each type of grammar entry is compiled into a DCG rule as follows.

Grammar Rules

All of the unification equations in the grammar rule are applied (Section 7.2.1), producing the feature structures associated with the rule. For example, the Lhs and Rhs variables of the rule

```
rule( VP, [ V ],[ [ VP, cat ] = vp,
                  [ V, cat ] = v,
                  [ VP, head ] = [ V, head ],
                  [ VP, subcat ] = [ V, subcat ] ] ).
```

become:

```
VP becomes [ cat: vp
             head: X
             subcat: Y ]

V becomes  [ cat: v
             head: X
             subcat: Y ]
```

These feature structures together with the rule itself are now compiled into a DCG rule, in left-corner format (Section 7.2.3).

At this point, the solution to the problem caused by epsilon rules is applied (Section 7.2.4). As a result, one rule may expand to a set of rules. These rules are written into the DCG file in a form that is slightly more complex than that presented in Section 7.2.3:

```
lc( Rhs1, Parent, Branch1, Tree ) -->
        down( Rhs2, Branch2 ), ... , down( RhsN, BranchN ),
        lc( Lhs, Parent, NewTree, Tree ).
```

where:

Rhs1... RhsN are the feature structures associated with the right-hand side of the rule.

Parent is the feature structure associated with the left-hand side of the rule.

Branch1 ... Branch2 are the parse trees associated with the right-hand side of the rule.

Tree is the parse tree associated with the left-hand side of the rule.

NewTree is the parse tree associated with the entire rule.

For example, the rule presented above becomes the DCG rule:

```
lc( [ cat: v,
      head: X,
      subcat: Y ],
    Parent, Branch1, Tree ) -->
lc( [ cat: vp,
      head: X,
      subcat: Y ],
    Parent, vp( Branch1 ), Tree ).
```

Lexical items

Each lexical item in the grammar is compiled into a DCG rule. These rules, unlike grammar rules, are not directly written into the DCG file. They are asserted into the database to be compiled and written into the DCG file in a later stage.

Each type of lexical item is asserted with a different functor but they are processed in the same way. First, all of the specifications in the definition are processed. If a specification is a unification, it is applied (Section 7.2.1); if it is a reference to a lexical rule or lexical template, the reference is put into the form

```
template( Name, In, Out )
        or
lex_rule( Name, In, Out )
```

where In is the input feature structure to a rule or template, and Out is the output feature structure from the rule or template. These references are expanded in the second compilation phase.

- *Lexical entries*

  Lexical entries are asserted into the database in the form

  ```
  word( Word, FeatureStructure ):-
      <references to rules and templates,
       producing FeatureStructure>.
  ```

  where Word is the name of a lexical entry, and FeatureStructure is the feature structure associated with the lexical entry.

  For example, the lexical entry

  ```
  lex( uther, [ [ lex = uther], [ sense = uther1 ], noun ] ).
  ```

  is compiled into

  ```
  word( uther, FeatureStructure ): -
      template( noun, [ lex: uther, sense: uther1 ],
                FeatureStructure ).
  ```

- *Lexical templates*

  Lexical templates are asserted into the database in the form

  ```
  template( Name, FeatureStructure ):-
      <references to templates and rules,
       producing FeatureStructure>.
  ```

where Name is the name of a lexical template, and FeatureStructure is the feature structure associated with the template.

For example, the lexical template

```
template( mainverb, [ [ head, aux = false], verb ] ).
```

is compiled into

```
template( mainverb, FeatureStructure ):-
        template( verb, [ head: [ aux: false ] ],
                FeatureStructure ).
```

- *Lexical rules*

  Unlike lexical entries and lexical templates, lexical rules are not allowed to contain references to rules or templates in their definition. Thus, lexical rules are asserted into the database in the form

  ```
  lex_rule( Name, In, Out ).
  ```

  where:
  Name is the name of a lexical rule.
  In is the feature structure associated with the input to the rule.
  Output is the feature structure associated with the output from the rule.

  For example, the lexical rule

  ```
  lex_rule( nom, [ [ Out, head ] = [ In, head ], [ Out, cat ] = n ] ).
  ```

  is compiled into

  ```
  lex_rule( nom, [ cat: v, head: X ], [ cat: n, head: X ] ).
  ```

Lexical compilation: Second stage

Lexical entries are initially compiled into DCG rules with explicit calls to the templates and lexical rules they utilize. Because these calls are reexecuted each time they are encountered, the system would be inefficient to use. At the second stage of compilation, these references are eliminated by merging the corresponding feature structures with the rest of the definition.

Once this process is completed, the DCG rules for the lexical entries no longer contain any references to any lexical templates or rules, therefore the rules and templates need not be recorded in the DCG file.

The new lexical entries are written into the DCG file as

```
lex( Word, FeatureStructure ).
```

For example, the initial DCG rules

```
word( boy, Y ):-
        template( noun, X, Y ).
template(noun, X, Y ):-
            Y = [ cat: n ].
```

produce the new DCG rule

```
lex( boy, [ cat: n ] ).
```

Sentence	Parse time (in seconds)
Uther sleeps	0.066
Uther storms Cornwall	0.067
Knights sleep	0.084
Cornwall is stormed	0.1
A knight storms Cornwall	0.1

Table 7.1: Execution Statistics

7.4 Conclusions

In order to test whether the P-PATR system lives up to the expectations that motivated its development, it will be necessary to compare it with the two other currently running PATR systems: D-PATR [5] and Z-PATR [18]. Due to disparities in the versions of the PATR formalism assumed by each system, at the present accurate statistics are not available, but the preliminary results seem promising.

Sample execution statistics can be seen in Table 7.1. These are the execution results from the DCG produced by P-PATR, using as input the grammar in Appendix C. From these statistics it is easy to see that a DCG produced by P-PATR is a speedy parsing tool.

7.4.1 Further Work

P-PATR is far from complete. Changes are being made to improve the performance of the system and to enhance its capabilities. These enhancements include:

- *Improved parser performance*

  Because Prolog uses a depth-first control strategy, a DCG produces the first parse for a sentence quickly, but when all parses must be produced, the necessary backtracking slows down the parse significantly. To solve this problem, predictive [9] capabilities will be added to P-PATR to eliminate some of the useless backtracking so that all parses are found more quickly.

- *Compatibility with the other PATR systems*

  To allow better performance comparisons, it would be desirable to be able to run the same grammar on P-PATR as on the other two systems discussed above. Some work is currently being done [16] on developing a standard specifying a specific version of the PATR formalism to which all PATR systems would conform. Once this is developed, it will be easy to use the same grammar on all PATR systems.

- *Morphological analysis*

  P-PATR currently does not perform morphological analysis. For each form of a lexical entry in a PATR grammar, a separate entry in the grammar must be present. By encorporating the work being done on morphological analysis in the PATR framework

[2] into P-PATR, only the stem forms of the lexical entries need to be entered in the lexicon.

Acknowledgments

I owe a great deal to many people, both for this document and for my mental well-being. Of course, thanks go to Lauri Karttunen. The fact that this document can be understood by anyone other than myself is due to his diligent dissection of the presentation. Thanks also go to Fernando Pereira, for his never-ending answers to my never-ending questions. His enthusiasm was quite infectious and kept me going when things looked bleak. I owe a lot to Ivan Sag and Carl Pollard, members of my moral support team, for keeping me from giving up in times of crisis.

However, the person to whom I owe the greatest thanks is Stuart Shieber, head of my moral support team. The existence of this document is to a large part due to his encouragement and technical guidance.

Finally I must thank my family, an important part of my life. The love and support of my parents is a constant comfort. My brother, Haym Hirsh, has also contributed to the completion of this work, from the pictures in the text to being on the other end of the phone when I call.

References

[1] Alfred Aho, Ravi Sethi, and Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley Publishing Co., 1986.

[2] John Bear. A morphological recognizer with syntactic and phonological rules. In *Proceedings of the Eleventh International Conference on Computational Linguistics.* University of Bonn, Bonn, German Federal Republic, 25-29 August 1986.

[3] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1981.

[4] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley Publishing Co., 1979.

[5] Lauri Karttunen. D-PATR: a development environment for unification-based grammars. CSLI Report No. CSLI-86-61, CSLI, Stanford, California, 1986.

[6] Yuji Matsumoto, Hozumi Tanaka, Hideki Hirakawa, Hideo Miyoshi, and Hideki Yasukawa. BUP: a bottom-up parser embedded in Prolog. *New Generation Computing,* 1:145–158, 1983.

[7] Fernando C. N. Pereira. Logic for natural language analysis. Technical Note 275, Artificial Intelligence Center, SRI International, Menlo Park, California, 1983.

[8] Fernando C. N. Pereira and David H. D. Warren. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics.* Massachusetts Institute of Technology, Cambridge, Massachusetts, 15-17 June 1983.

[9] Fernando C.N. Pereira. Deductive computation of grammar properties. Forthcoming.

[10] Fernando C.N. Pereira and Stuart M. Shieber. *Prolog and Natural Language Analysis.* CSLI, Stanford, California, 1987. Forthcoming.

[11] D.J. Rosenkrantz and P.M. Lewis II. Deterministic left corner parsing. In *IEEE Conference Record 11th Annual Symposium on Switching and Automata Theory,* 1968.

[12] Stuart M. Shieber. Criteria for designing computer facilities for linguistic analysis. *Linguistics,* 23:189–211, 1985.

[13] Stuart M. Shieber. The design of a computer language for linguistic information. In *Proceedings of the Tenth International Conference on Computational Linguistics,* Stanford University, Stanford, California, 2-7 July 1984.

[14] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar.* CSLI, Stanford, California, 1986.

[15] Stuart M. Shieber. The PATR-II experimental system. 1984. Stanford University, Stanford, California.

[16] Stuart M. Shieber. Standard for the PATR computer language. Forthcoming.

[17] Stuart M. Shieber, Lauri Karttunen, and Fernando C. N. Pereira. *Notes from the Unification Underground: A Compilation of Papers on Unification-Based Grammar Formalisms.* Technical Report 327, Artificial Intelligence Center, SRI International, Menlo Park, California, June 1984.

[18] Stuart M. Shieber, Hans Uszkoreit, Fernando C. N. Pereira, Jane J. Robinson, and Mabry Tyson. The formalism and implementation of PATR-II. In *Research on Interactive Acquisition and Use of Knowledge*, Artificial Intelligence Center, SRI International, Menlo Park, California, 1983.

[19] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques.* The MIT Press, Cambridge, Massachusetts, 1986.

[20] David Warren, Luis Pereira, and Fernando Pereira. Prolog - the language and its implementation compared with Lisp. In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, University of Rochester, Rochester, New York, 15-17 August 1977.

Chapter 8

Towards a Deductive Theory of Sentence Interpretation

This chapter was written by Fernando C. N. Pereira

Abstract

This chapter introduces a deductive view of the sentence interpretation process in natural language based on the notion of conditional interpretation. The main goal of this new framework is to provide a uniform treatment of interactions between semantic and pragmatic phenomena, in particular quantification and anaphora, which have caused difficulties to existing compositional approaches. The chapter covers theoretical motivation and basic concepts. Details of a first implementation of the theory are covered in a companion chapter [19].

8.1 The Breakdown of Compositionality

Logic is widely employed in theories of natural language. In syntax, logic and unification grammars [14,24] are particular instances of a view of grammar as a formal axiomatic theory, grammaticality judgements as consequences of the theory, analysis as proofs, and parsing as proof construction. The slogan *parsing as deduction* [21,12] has been used as a shorthand for this deductive view of grammar and parsing. In semantics, going back to logic's origins as an abstraction and formalization of natural-language arguments, logical systems have been used to give rigorous accounts of sentence meaning. In this chapter, however, I am concerned with yet another use of logic in a theory of natural language, namely its use in a deductive model of the sentence interpretation process.

Compositional semantics, developed by Montague and his followers, brought for the first time to theories of natural language the kind of rigor imparted to formal logic by Tarskian truth definitions. It provides a precise theory of interpretation, in which syntactic com-

position functions (or syntactic rules) that assemble phrases into larger phrases are paired with semantic functions that map the denotations[1] of the subphrases into the denotation of the whole phrase. Unfortunately, the compositional description of the relation between utterances and their meanings for any substantial fragment of English requires encoding into the denotation of an expression not only its meaning but also information about the expression's internal structure and context of use. A good example of this is the Cooper storage mechanism for quantifier scoping [3]. A similar problem arises in the denotational semantics of imperative programming languages, where the denotation of an expression must encode information about the occurrence of the expression (eg. continuations) [25]. This problem was one of the motivations for the development of the alternative theory of *abstract operational semantics* of programming languages [22]. Not coincidentally, the techniques I will use to deal with the limitations of strict compositionality in natural-language semantics have some similarities to those of abstract operational semantics.

There are by now a set of standard techniques in computational linguistics to loosen the compositional straitjacket. The basic move is to abandon the direct translation of phrases to their (model-theoretic) meanings via semantic functions and instead translate phrases to formal intermediate expressions that are then combined in such a way that the formal expression translating a sentence, the sentence's *logical form*, is a closed formula of some appropriate logical language. It is then possible for the semantic functions associated to syntactic rules to be sensitive to the forms of the translations of phrases rather than just to their denotations [30,7,27,23]. [2]

In Montague grammar, ambiguities such as that of quantifier scope are relegated to the realm of syntax: syntactic rules must generate alternative analyses for ambiguous sentences, since the functional nature of semantic rules prevents the generation of more than one translation for a given analysis. As more types of semantic ambiguity are considered, this approach becomes untenable, requiring the syntactic rules to generate alternative analyses that have no syntactic import whatsoever.

An alternative approach to the representation of semantic ambiguity is to use semantic *relations* instead of semantic functions. In the interests of modularity, it is often convenient to factor those semantic relations as the composition of a semantic function, associated to a particular syntactic rule, and a relation that generates semantic alternatives independently of particular syntactic rules. Quantifier scope and anaphora have often been treated in this way [30,6], to the point of dividing the interpretation process into two phases: sentences are translated first into unscoped formal expressions (similar to matrix-store pairs in Cooper's account), which are then given to a scope determination algorithm that computes the relation between unscoped forms and final logical forms [29,23,11]. An important question in the formulation of semantic relations is whether the generation of semantic alternatives is *incremental* in the sense that the alternative translations for a phrase are constructed

[1] I will use the term "denotation" to refer to the mathematical entities into which phrases are mapped by the semantic interpretation process, and which purport to represent rigorously the meaning of those phrases.

[2] Logical forms have also been used in Montague's PTQ [18] and related work, but in this case they are just a convenient representation for the denotations of phrases in translation functions. That is, the denotation of the result of a translation function does not depend on the logical forms of its arguments, but only on their denotations.

by combining compatible translations for its constituents. For instance, quantifier scoping mechanisms based on Cooper storage [3] or on its computational analogues satisfy this requirement, but Montague grammar and discourse-representation theory (DRT) [13,9], discussed below, do not.

In DRT, an intermediate level of discourse-representation structures (DRSs) is the output language for the translation process. The binding mechanisms in this intermediate level allow a more uniform treatment of the interactions between indefinite noun phrases and anaphora, as in the celebrated "donkey sentences" [5]. DRS-construction rules are usually given formally, that is, in terms of the forms of the DRSs being combined, but DRSs form a logical language with an inductive truth definition analogous to the one for the predicate calculus. DRS-construction rules can only be applied once the relative scopes of noun phrases and anaphoric bindings have been determined, making the construction process nonincremental.

Strict compositionality makes semantic rules very simple: they just state which semantic combination function corresponds to each syntactic combination rule. However, as the foregoing discussion indicates, semantic composition rules for more extensive natural-language fragments are much more delicate. The rules become then an important object of study in themselves. But none of the approaches above says anything explicit about the nature of semantic rules – what objects they operate upon and what operations they are allowed to perform on those objects. For instance, rules dealing with the interactions between anaphora and quantification involve conditions at different descriptive levels, from discourse context to constraints on binding in the output logical form [28]. Typically, such rules have been encoded in terms of arbitrary formal operations on concrete representations (eg. list structure) of the various objects involved. Besides being a programming nightmare because of their lack of abstraction, such "ad hoc" rules reflect a complete abandonment of compositionality.

8.2 Conditional Interpretations

The argument of the preceding section leads to the question of whether there is a intermediate position between strict compositionality and total arbitrariness in semantic rules. I will argue that such middle ground can indeed be found.

The difficulties of strict compositionality are caused by lack of information: we are rarely in a position to give a closed-form meaning for an expression without further information about its context of use. However, we can often give phrases a *conditional* interpretation: phrase p has interpretation ϕ provided that assumptions Γ on p's use are satisfied, in symbols $\Gamma \vdash p \rightsquigarrow \phi$. In general, the logical form ϕ will be *dependent* on Γ (through shared parameters), so satisfying (or *discharging*) the assumptions Γ may further specify ϕ. I will assume that all parameters free in ϕ occur free in Γ, and I will denote by $\nu(\Gamma)$ the set of free parameters in Γ.

For example, we might have

$$\mathsf{bind}(\mathrm{the}', x, \mathrm{dog}') \vdash \text{“the dog sleeps”} \rightsquigarrow \mathrm{sleeps}'(x) \tag{8.1}$$

where the assumption $\mathbf{bind}(\text{the}', x, \text{dog}')$ requires the availability of a contextually unique entity x of type "dog".

It is worth relating this notion of conditional interpretation to the relational theory of meaning from situation semantics [2]. According to that theory, the meaning of a sentence[3] s is a relation $[\![s]\!]$ between situations u in which s is uttered and situations d correctly described by s being uttered in u. A logical form can be taken as giving a [parameterized] condition on described situations, and a set of assumptions a [parameterized] condition on utterance situations. Using the notation $t \models_a \gamma$ to say that situation t satisfies condition γ under the anchor a for the parameters occurring in γ, the interpretation judgement $\Gamma \vdash s \rightsquigarrow \phi$ should correspond to the relationship $u[\![s]\!]d$ holding for all utterance situations u, described situations d and anchors a for the parameters $\nu(\Gamma)$ such that $d \models_a \phi$ whenever $u \models_a \Gamma$ and s is uttered in u.

A conditional interpretation $\Gamma \vdash s \rightsquigarrow \phi$ may thus be understood as a representation of what can be determined about the situation described by s just with the information that s was uttered. The fact that s was uttered in situation u imposes additional constraints on u, represented by the assumptions Γ. Partial execution in logic programming [26,20] provides a good analogy here: the definition of the ternary relation $u[\![s]\!]d$ for any u and d is partially expanded to determine d for a given s but unknown u. The logical form ϕ represents the answer, and the assumptions Γ represent constraints that cannot be solved without information about u.

The preceding discussion illustrates well that the logical form ϕ in a conditional interpretation $\Gamma \vdash s \rightsquigarrow \phi$ does not represent the *meaning* of s in situation-semantics terms, but rather a constraint of the possible *interpretations* of s. For a given anchor a of $\nu(\Gamma)$, the described situations d such that $d \models_a \phi$ form the interpretation, in the sense of situation-semantics, of any utterance situation u such that s is uttered in u and $u \models_a \Gamma$. That is, ϕ can be seen as [a representation of] a parameterized interpretation of s, which yields a specific interpretation for each choice of an utterance situation satisfying the conditions Γ.

This explanation of conditional interpretations in terms of the relational theory of meaning gives an adequate account of conditional interpretations such as (8.1), but it will require further work to extend it to accomodate the full range of formal assumptions used in the semantic rules in this chapter, in particular those created by general rather than singular noun phrases.

8.3 Composing Interpretations

I have just introduced conditional interpretations as a means of factoring out the context-dependent aspects of phrase interpretation. However, we also need to know how conditional interpretations of phrases are build from the conditional interpretations for their constituents. The basic idea is extremely simple, and formally reminiscent of sequent-calculus proof systems.

[3] I mean here by "sentence" a sentence type, rather than a sentence token, which is better called an "utterance."

We will have two types of semantic interpretation rules: *structural* rules, which are associated to a specific syntactic rule, and *discharge* rules, which can apply whenever assumptions fit a certain pattern.

Assumptions and interpretations in rules and derivations will be expressed in the language of the typed λ-calculus, although the actual types of terms will be mostly left implicit. In particular, the interpretation judgement operator $\rightsquigarrow$ is overloaded, since strictly speaking there should be distinct operators $\Gamma \rightsquigarrow_\tau \phi$ corresponding to the different types τ of the interpretation ϕ: individuals (ι), propositions (o) and predicates ($\iota \to o$), among others. This distinction will be important in some of the discharge rules, which only apply to interpretations of certain types.

8.3.1 Structural Rules

Structural rules have the general form

$$\frac{\Gamma_1 \vdash p_1 \rightsquigarrow \phi_1 \cdots \Gamma_n \vdash p_n \rightsquigarrow \phi_n}{\Gamma \vdash \mathbf{R}(p_1, \ldots, p_n) \rightsquigarrow \phi} \tag{8.2}$$

where $\mathbf{R}$ represents a syntactic rule. A rule may have extra conditions that restrict its applicability, for instance conditions to avoid free-variable capture by binding operators.

For example, for the syntactic rule $R_{\mathrm{S}\to\mathrm{NP\ VP}}$ that combines a noun phrase with a verb phrase to make a sentence, we could have

$$\frac{\Gamma \vdash N \rightsquigarrow n \qquad \Delta \vdash V \rightsquigarrow v}{\Gamma, \Delta \vdash R_{\mathrm{S}\to\mathrm{NP\ VP}}(N, V) \rightsquigarrow v(n)} \quad \textsf{[sent]}$$

For the syntactic rule $R_{\mathrm{NP}\to\mathrm{Det\ Nom}}$ that combines a determiner with a nominal to make a noun phrase, we could have

$$\frac{\Gamma \vdash D \rightsquigarrow d \qquad \Delta \vdash N \rightsquigarrow n}{\Gamma, \Delta, \mathsf{bind}(d, x, n) \vdash R_{\mathrm{NP}\to\mathrm{Det\ Nom}}(D, N) \rightsquigarrow x} \quad \textsf{[np]}$$

where x does not occur free in Γ, Δ, d or n. Finally, the following rule corresponds to the recategorization of an intransitive verb phrase as a verb phrase:

$$\frac{\Gamma \vdash V \rightsquigarrow v}{\Gamma \vdash R_{\mathrm{VP}\to\mathrm{IV}}(V) \rightsquigarrow v} \quad \textsf{[ivp]}$$

Lexical items are introduced by *axioms*, rules with empty antecedents, keyed by lexical category rather than by syntactic rules. All lexical rules that we will need are instances of the following schema:

$$\frac{}{\vdash C(l) \rightsquigarrow_{\tau_C} l'} \quad \textsf{[lex]}$$

where C is a lexical category, l' is the sense of lexical item l, and τ_C is the type of the sense of any lexical item of category C. The following assignments are for now sufficient:

Det (determiner)	$(\iota \to o) \to (\iota \to o) \to o$
N (noun)	$\iota \to o$
N_2 (relational noun)	$\iota \to \iota \to o$
IV (intransitive verb)	$\iota \to o$
TV (transitive verb)	$\iota \to \iota \to o$

These assignments are very close to those used in compositional accounts [4], except that the assumption mechanism makes it unnecessary to raise the type of the object of a transitive verb from ι to $(\iota \to o) \to o$, as was done by Montague [18].

Given the syntactic analysis

$$[_S[_{NP}[_{Det}\text{the}][_N\text{dog}]][_{VP}[_{IV}\text{sleeps}]]] \quad ,$$

for "the dog sleeps", its interpretation (8.1) would be constructed by the following derivation:

$$\dfrac{\dfrac{\dfrac{}{\vdash \text{“the”} \rightsquigarrow \text{the}'}\,[\text{lex}] \quad \dfrac{}{\vdash \text{“dog”} \rightsquigarrow \text{dog}'}\,[\text{lex}]}{\text{bind}(\text{the}', x, \text{dog}') \vdash \text{“the dog”} \rightsquigarrow x}\,[\text{np}] \quad \dfrac{\dfrac{}{\vdash \text{“sleeps”} \rightsquigarrow \text{sleeps}'}\,[\text{lex}]}{\vdash \text{“sleeps”} \rightsquigarrow \text{sleeps}'}\,[\text{ivp}]}{\text{bind}(\text{the}', x, \text{dog}') \vdash \text{“the dog sleeps”} \rightsquigarrow \text{sleeps}'(x)}\,[\text{sent}]$$

Two other structural rules will be needed for the exemples that follow. The first goes with the syntactic rule that combines a transitive verb with its object:

$$\dfrac{\Gamma \vdash V \rightsquigarrow v \qquad \Delta \vdash N \rightsquigarrow n}{\Gamma, \Delta \vdash R_{\text{VP}\to\text{TV NP}}(V, N) \rightsquigarrow \lambda x.v(x, n)} \qquad [\text{tvp}]$$

The second is rather similar, and deals with relational nouns:

$$\dfrac{\Gamma \vdash N \rightsquigarrow n \qquad \Delta \vdash N' \rightsquigarrow n'}{\Gamma, \Delta \vdash R_{\text{Nom}\to\text{N}_2\text{ of NP}}(N, N') \rightsquigarrow \lambda x.n(x, n')} \qquad [\text{reln}]$$

where x is assumed not to occur free in v, n or n'.

8.3.2 Discharge Rules

While structural rules give the conditional interpretation of particular syntactic constructions, discharge rules are used to eliminate (*discharge*) or modify assumptions at any point in a derivation in which their requirements are satisfied. Structural rules are obligatory in the sense that some structural rule associated with a given syntactic rule must be applied to any phrase analyzed by the syntactic rule, whereas discharge rules are optional in that they do not need to be applied even at points in a derivation where their requirements are satisfied[4]. For example, by applying discharge rules at different points in alternative

[4]However, not applying a discharge rule at some point in a derivation being constructed may prevent the latter application of other rules, and lead the [partial] derivation to a dead end.

derivations for the same sentence, alternative dependencies between noun phrases (scopings and anaphoric links) are produced.

Discharge rules used in this chapter have the general form

$$\frac{\Gamma \vdash p \rightsquigarrow_{\alpha} \phi}{\Delta \vdash p \rightsquigarrow_{\beta} \psi} \tag{8.3}$$

where the type subscripts α and β indicate the applicability of the rule following the convention stated earlier.

The following rule allows a **bind** assumption to be discharged by applying its determiner as a generalized quantifier [1] to a property, establishing the scope of the quantifier:

$$\frac{\mathsf{bind}(Q, x, R), \Gamma \vdash p \rightsquigarrow_{\iota \to o} T}{\Gamma \vdash p \rightsquigarrow_{\iota \to o} \lambda y.Q(R, \lambda x.T(y))} \quad \textsf{[qpred]}$$

where y does not occur free in Q, R or T and x does not occur free in Γ. The following derivation shows how the rule works:

$$\frac{\mathsf{bind}(\text{every}', c, \text{child}') \vdash \text{“friend of every child”} \rightsquigarrow \lambda f.\text{friend}'(f, c)}{\vdash \text{“friend of every child”} \rightsquigarrow \lambda f.\text{every}'(\text{child}, \lambda c.\text{friend}'(f, c))} \textsf{[qpred]}$$

A sentence-level version of rule **[qpred]** can also be used to discharge **bind** assumptions:

$$\frac{\mathsf{bind}(Q, x, R), \Gamma \vdash p \rightsquigarrow_{o} S}{\Gamma \vdash p \rightsquigarrow_{o} Q(R, \lambda x.S)} \quad \textsf{[qprop]}$$

where x does not occur free in Γ.

Rules **[qpred]** and **[qprop]** are essential to generate alternative scopings for the quantifiers in the interpretation of a sentence. Figures 8.1 and 8.2 show derivations giving alternative scopes for “a friend of every child came.” Instead of the full derivation format of the preceding examples, those derivations are given in a simplified form in which each node of a syntax tree is annotated with a sequence of conditional interpretations: the first one the result of applying the structural rule corresponding to the node, and the others the result of applying some discharge rule to the previous interpretation in the sequence.

8.4 Capture

The interaction between determiner and nominal assumptions in a noun phrase is not completely described by the **[np]** and **[qpred]** rules above. This question arises in particular in the interpretation of “donkey sentences” such as “every owner of a donkey beats it” [5]. I will discuss below a *capture* mechanism account for dependencies between quantified noun phrases and singular terms in such sentences. I do not claim that capture provides a definitive account of donkey sentences, but only that it exemplifies how conditional interpretations and discharge rules can be used to describe interactions between different binding mechanisms in sentences.

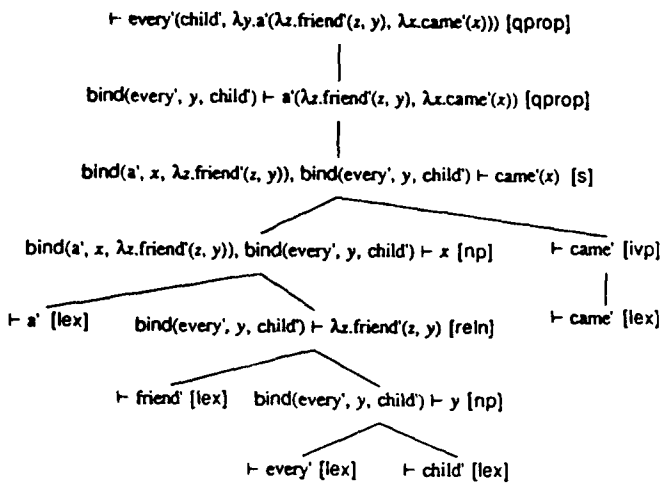


Figure 8.1: Wide Scope Universal

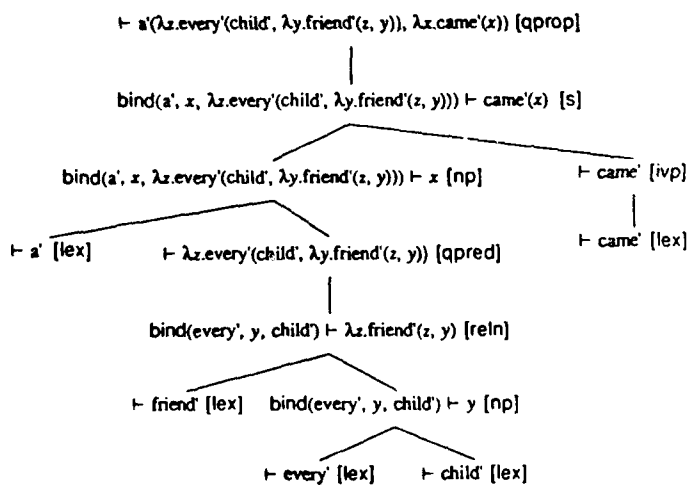


Figure 8.2: Narrow Scope Universal

When rule [np] is applied to a complex noun phrase, the sort term n corresponding to the interpretation of the nominal part of the noun phrase will in general contain free parameters bound by assumptions Δ. Side conditions in rules [qpred] and [qprop] ensure that those parameters are properly bound in the final interpretation.[5]

The capture rule will require further side conditions, for which we need some auxiliary concepts. An assumption $\mathsf{bind}(d, x, n)$ *depends* on another assumption $\mathsf{bind}(d', x', n')$ if x' is free in n. Quantifiers (the meanings of determiners) will be classified as either *general* (eg. every′, most′) or *singular* (eg. a′, the′). General quantifiers will be allowed to capture singular quantifiers on which they depend.

With the above definitions, the capture rule is simply

$$\frac{\mathsf{bind}(S, x, p), \mathsf{bind}(G, y, q), \Gamma \vdash r \rightsquigarrow_o R}{\mathsf{bind}(\forall, x, p), \mathsf{bind}(G, y, q), \Gamma \vdash r \rightsquigarrow_o R} \qquad \text{[capt]}$$

where $\mathsf{bind}(G, y, q)$ depends on $\mathsf{bind}(S, x, p)$, G is general, S is singular and $\forall$ is the first-order universal quantifier.

To show the effect of this rule in donkey sentences, we will also need rules for intrasentential anaphora. The actual system based on the present theory includes a substantial account of discourse and other constraints on anaphora [19]. Those issues, however, are beyond the scope of this chapter, so I will use some rather simplistic rules for anaphora. The personal pronoun "it" is introduced by the axiom

$$\frac{}{\mathsf{bind}(\mathsf{it}, x, \lambda t.\mathsf{true}) \vdash \text{“it”} \rightsquigarrow x} \qquad \text{[it]}$$

Ignoring coreference constraints, intrasentential anaphora involves the following discharge rule

$$\frac{\mathsf{bind}(Q, x, p), \mathsf{bind}(\mathsf{it}, y, \lambda t.\mathsf{true}), \Gamma \vdash S \rightsquigarrow s}{\mathsf{bind}(Q, x, p), \Gamma \vdash S \rightsquigarrow (\lambda y.s)(x)} \qquad \text{[bind]}$$

Figure 8.3 shows the application of the rules so far to produce the interpretation

$$\forall(\text{machine}', \lambda m.\text{most}'(\lambda u.\text{user}'(u, m), \lambda xy.\text{breaks}'(x, y))) \tag{8.4}$$

for the sentence "most users of a machine break it." This seems the "strongest" (with fewest models) reasonable interpretation for the sentence. If the application of the capture rule were removed from the derivation, the resulting interpretation would be the alternative one in which some machine is broken by most of its users:

$$\text{a}'(\text{machine}', \lambda m.\text{most}'(\lambda u.\text{user}'(u, m), \lambda xy.\text{breaks}'(x, y))) \tag{8.5}$$

which (for a nonempty set of machines) is weaker than (8.4).[6]

[5] Side conditions are a rather unsatisfactory way of representing dependencies. An explicit encoding of dependencies, along the lines of the Logical Framework [10], might be better.

[6] Other plausible interpretations, also weaker than (8.4) could be constructed by alternative capture rules introducing choice functions mapping users to machines they use.

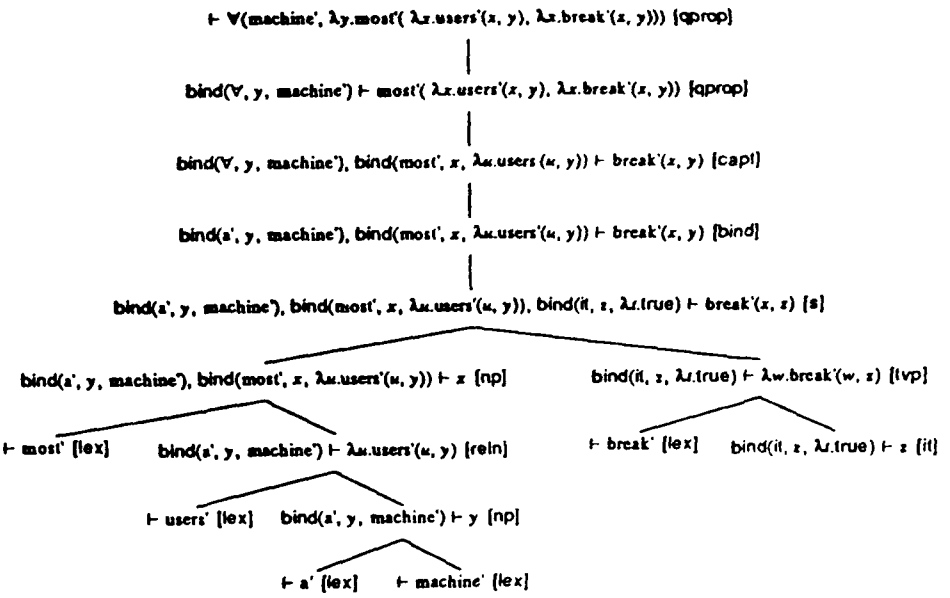


Figure 8.3: Donkey Sentence

8.5 Conclusion and Further Work

In this chapter I have tried to show how a deductive framework can be used to keep the construction of sentence interpretations compositional, by separating out in assumptions the manipulations on bindings and scope that cause difficulties for standard compositional accounts. The process of phrase interpretation is identified with the construction of a derivation of a judgement that the phrase has some interpretation under certain assumptions, where derivations are the result of applying appropriate rules of inference. The main current weakness of the approach is the lack of a full justification for the concept of conditional interpretation and for specific interpretation rules in terms of a semantic theory independent of the interpretation process. The relational theory of meaning provides a partial justification for conditional interpretations for assumptions introduced by singular terms, but it is not yet clear how this justification will have to be modified to cover the assumptions introduced by general noun phrases as well as other kinds of assumptions not discussed here but required to cover the wider range of constructions covered in the implemented [19].

That system deals with various discourse mechanisms supporting constraints in anaphora and definite reference, which require a more complex interaction between assumptions and discourse context than I have used here. Rules must access a representation of the discourse context and produce not only a conditional interpretation for a phrase but also a revised discourse context. There is a strong similarity between rules of this kind and

abstract-operational-semantics rules as described by Plotkin [22].

As noted before, side conditions are a rather messy way of enforcing constraints on binding in rules. The Logical Framework [10] uses a theory of higher-order types and functions to represent proof rules and systems so that side conditions are replaced by the standard provision against variable clashes in β-reduction. Similar effects are achieved in λ-Prolog [15,16] by using higher-order unification and conditional goals with locally-bound variables [17]. A reformulation of the present rule set in one of these frameworks would be worthwhile.

Acknowledgments

I thank Jan van Eijck and Martha Pollack for many useful comments and corrections. All the remaining errors are of course my own. The ideas in this chapter are based on earlier joint work with Martha Pollack.

References

[1] J. Barwise and R. Cooper. Generalized quantifiers and natural language. *Linguistics and Philosophy*, 4:159–219, 1981.

[2] J. Barwise and J. Perry. *Situations and Attitudes.* MIT Press, Cambridge, Massachusetts, 1983.

[3] R. Cooper. *Quantification and Syntactic Theory.* Volume 21 of *Synthese Language Library*, D. Reidel, Dordrecht, Netherlands, 1983.

[4] D. R. Dowty, R. E. Wall, and S. Peters. *Introduction to Montague Semantics.* Volume 11 of *Synthese Language Library*, D. Reidel, Dordrecht, Netherlands, 1981.

[5] P. T. Geach. *Reference and Generality.* Cornell University Press, Ithaca, New York, 1962.

[6] B. Grosz, D. E. Appelt, P. Martin, and F. Pereira. TEAM: an experiment in the design of transportable natural language interfaces. *Artificial Intelligence*, 31, 1987.

[7] B. Grosz, N. Haas, G. G. Hendrix, J. Hobbs, P. Martin, R. Moore, J. Robinson, and S. Rosenschein. *DIALOGIC: A Core Natural-Language Processing System.* Technical Note 270, Artificial Intelligence Center, SRI International, Menlo Park, California, November 1982.

[8] B. J. Grosz, K. S. Jones, and B. L. Webber, editors. *Readings in Natural Language Processing.* Morgan Kaufmann, Los Altos, California, 1986.

[9] F. Guenthner, H. Lehmann, and W. Schönfeld. A theory for the representation of knowledge. *IBM Journal of Research and Development*, 30(1):39–56, January 1986.

[10] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proceedings of the Second Symposium on Logic in Computer Science*, Cornell University, IEEE, Ithaca, New York, 1987.

[11] J. R. Hobbs and S. M. Shieber. An algorithm for generating quantifier scopings. *Computational Linguistics*, 13, 1987.

[12] M. E. Johnson. *Attribute-Value Logic and the Theory of Grammar*. PhD thesis, Department of Linguistics, Stanford University, Stanford, California, 1987.

[13] H. Kamp. A theory of truth and semantic interpretation. In J. A. G. Groenendijk, T. M. V. Janssen, and M. B. J. Stokhof, editors, *Formal Methods in the Study of Language*, pages 277–322, Mathematisch Centrum, Amsterdam, Netherlands, 1981.

[14] M. Kay. Parsing in functional unification grammar. In D. R. Dowty, L. Karttunen, and A. M. Zwicky, editors, *Natural Language Parsing*, chapter 7, pages 251–278, Cambridge University Press, Cambridge, England, 1985.

[15] D. A. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *Third International Conference on Logic Programming*, Springer-Verlag, Berlin, Germany, 1986.

[16] D. A. Miller and G. Nadathur. Some uses of higher-order logic in computational linguistics. In *Proceedings of the 24th Annual Meeting*, pages 247–256, Association for Computational Linguistics, Columbia University, New York, 1986.

[17] D. A. Miller, G. Nadathur, and A. Scedrov. Hereditary harrop formulas and uniform proof systems. In *Proceedings of the Second Symposium on Logic in Computer Science*, Cornell University, IEEE, Ithaca, New York, 1987.

[18] R. Montague. The proper treatment of quantification in ordinary English. In R. H. Thomason, editor, *Formal Philosphy*, Yale University Press, 1973.

[19] F. C. N. Pereira and M. E. Pollack. An integrated framework for semantic and pragmatic interpretation. November 1987. In preparation.

[20] F. C. N. Pereira and S. M. Shieber. *Prolog and Natural-Language Analysis*. Volume 10 of *CSLI Lecture Notes*, Center for the Study of Language and Information, Stanford, California, 1985. Distributed by Chicago University Press.

[21] F. C. N. Pereira and D. H. D. Warren. Parsing as deduction. In *Proceedings of the 21st Annual Meeting*, Association for Computational Linguistics, Cambridge, Massachusetts, June 15-17 1983.

[22] G. D. Plotkin. *A Structural Approach to Operational Semantics*. Lecture notes DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981.

[23] S. J. Rosenschein and S. M. Shieber. Translating English into logical form. In *Proceedings of the 20th Annual Meeting*, pages 1–8, Association for Computational Linguistics, University of Toronto, Toronto, Canada, 1982.

[24] S. M. Shieber. *An Introduction to Unification-Based Approaches to Grammar.* Volume 4 of *CSLI Lecture Notes*, Center for the Study of Language and Information, Stanford, California, 1985. Distributed by Chicago University Press.

[25] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Massachusetts, 1977.

[26] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Proceedings of the Second International Logic Programming Conference*, pages 127–138, Uppsala University, Uppsala, Sweden, July 1984.

[27] D. H. D. Warren and F. C. N. Pereira. An efficient easily adaptable system for interpreting natural language queries. *American Journal of Computational Linguistics*, 8(3-4):110–122, July 1982.

[28] B. Webber. So what can we talk about now? In M. Brady and R. Berwick, editors, *Computational Models of Discourse*, pages 331–371, MIT Press, Cambridge, Massachusetts, 1983.

[29] W. A. Woods. *Semantics and Quantification in Natural Language Question Answering.* Report 3687, Bolt Beranek and Newman Inc., November 1977. Reprinted in [8].

[30] W. A. Woods, R. M. Kaplan, and B. Nash-Webber. *The Lunar Sciences Natural Language Information System: Final Report.* Report 3438, Bolt Beranek and Newman Inc., June 1972.

Chapter 9

An Integrated Framework for Semantic and Pragmatic Analysis

This chapter was written by Martha Pollack and Fernando Pereira.

Abstract

We report on a mechanism for semantic and pragmatic interpretation that has been designed to take advantage of the generally compositional nature of semantic analysis, without unduly constraining the order in which pragmatic decisions are made. To achieve this goal, we introduce the idea of a *conditional interpretation*: one that depends upon a set of assumptions about subsequent pragmatic processing. Conditional interpretations are constructed compositionlly according to a set of declaratively specified interpretation rules. The mechanism can handle a wide range of pragmatic phenomena and their interactions.

9.1 Introduction

Compositional systems of semantic interpretation, while logically and computationally very attractive [6,20,26], seem unable to cope with the fact that the syntactic form of an utterance is not the only source of information about its meaning. Contextual information—information about the world and about the history of a discourse—influences not only an utterance's meaning, but even its preferred syntactic analysis [3,5,7,16]. Of course, context also influences the interpretation (or meaning in context) of the utterance, in which, for example, referring expressions have been resolved.

One possible solution is to move to an integrated system of semantic and pragmatic interpretation, defined recursively on syntactic analyses that are neutral about those decisions that depend upon context. In this approach, a *least-commitment grammar* may be used to produce neutral representations that can be reconfigured later. Such a grammar might, for example, leave quantifiers in place [30], attach all prepositional phrases low and right [22],

and bracket to the right all compound nominals.[1] These neutral analyses can then serve as input to a system that produces interpretations (and not meanings) in a nearly compositional manner, in that the interpretation of a phrase[2] is a function of the interpretations of its syntactic constituents together with its context of utterance.

This model of semantic interpretation assumes that contextual information is available whenever it is needed for deciding among alternative interpretations. However, this is often not the case: questions about the interpretation of some constituent of an utterance might be answerable only when information about the interpretation of syntactically distant constituents becomes available. Familiar examples of this can be found, for instance, in sentences with quantifier scoping ambiguities and in sentences that include intrasentential anaphora. The so-called donkey sentences [9] exhibit both these phenomena.

These difficulties do not necessitate a complete abandonment of compositionality. To take advantage of the generally compositional nature of semantic analysis without constraining unduly the order in which pragmatic decisions are made, we assign to phrases *conditional interpretations*, which represent the dependence of a phrase's interpretation on assumptions about subsequent pragmatic processing. Conditional interpretations are built compositionally according to declaratively specified interpretation rules.

The interpretation mechanism we discuss here has been implemented in Prolog as part of the Candide system, a multimodal tool for knowledge acquisition. Incorporating both a graphical interface and a processor for English discourse, Candide allows a user of the Procedural Reasoning System (PRS) [10] to build and maintain procedural networks in a natural way. Procedural networks, an essential part of PRS's knowledge base, encode the information about procedures that is used by PRS for reasoning about and performing tasks in any given domain. The current version of Candide has been used to construct networks for malfunction procedures for NASA's space shuttle. Further details of the Candide system will be presented elsewhere [24].

9.2 Conditional Inter–pre–ta–tions

In our approach to semantic and pragmatic interpretation, conditional interpretations separate the context-independent aspects of an interpretation from those that are context-dependent. Each conditional interpretation consists of a *sense* and a [possibly empty] set of *assumptions*. As a first approximation, one might think of the sense of a phrase as representing purely semantic information, that is, information that can be adduced solely from the linguistic content of the phrase, no matter in which context the phrase has been uttered. The assumptions then represent constraints relating the phrase's sense to its ultimate interpretation. A *complete* interpretation has an empty assumption set, indicating that all of its dependencies on context have been resolved.

[1] There are reasons to suspect that ultimately syntactic analysis should be incorporated into the same stage of processing as semantic and pragmatic analysis; in particular, it is difficult to develop syntactically neutral representations for certain constructions such as conjunction.

[2] For simplicity, we shall use the term "phrase" to refer both to an entire utterance and to a constituent of an utterance, distinguishing between the two only when needed.

The present version of the theory allows for two kinds of assumptions. A *bind assumption* introduces a new parameter in an interpretation and places constraints on the binding of the parameter to individuals in the context. A *restrict assumption* does not introduce a new parameter, but instead further restricts the way in which an existing parameter can be bound.

These concepts are illustrated by the following conditional interpretation of the sentence "The jet failed":

$$[\![\text{"The jet failed"}]\!] = \langle \mathit{fail}(x), \{\mathsf{bind}(x, \mathsf{def}, \mathit{jet})\}\rangle \tag{9.1}$$

The first element of the interpretation is the sense $\mathit{fail}(x)$, while the second is the set of assumptions containing a single assumption whose informal reading is that x should be bound to something of the sort *jet* according to the constraints of definite reference.

9.3 The Interpretation Process

The process of semantic and pragmatic interpretation computes complete interpretations of sentences from least-commitment parse trees. Two types of rules govern the interpretation process: *semantic-interpretation ,rules* and *pragmatic-discharge rules.*

Semantic-interpretation rules specify the conditional interpretation of a phrase in terms of the conditional interpretations of its constituents. Compositionality is enforced by making semantic-interpretation rules sensitive only to the syntactic types of a phrase and its constituents, as well as to the types of assumptions in the conditional interpretations associated with the constituents; semantic-interpretation rules are *not* sensitive to the senses of the constituents.

Pragmatic-discharge rules change the conditional interpretation of a phrase by specifying how assumptions in the conditional interpretation may be eliminated with respect to the context of utterance. For example, one discharge rule applies to assumptions constraining a parameter to be bound as a definite reference. This rule allows an assumption of the form $\mathsf{bind}(v, \mathsf{def}, T)$ to be discharged, provided that there is a unique contextually available entity of sort T. The effect of applying the definite discharge rule to an interpretation $\langle S, A\rangle$ is twofold: the bind assumption operated upon is removed from the set of assumptions A; the sense S is changed to reflect the binding. For instance, if the rule were applied to the interpretation in (9.1), and if the context of utterance C contained a unique available entity j of sort *jet*, the resulting interpretation would be

$$\langle \mathit{fail}(j), \phi\rangle \tag{9.2}$$

As we shall see in the next section, assumption discharge will in general not only make use of but also change the discourse context. Therefore, discharge rules should be viewed as four-place relations. For example, the following would be an instance of the discharge relation:

$$\mathsf{discharge}(C, \langle \mathit{fail}(x), \{\mathit{bind}(x, \mathsf{def}, \mathit{jet})\}\rangle,$$

$$\langle fail(j), \phi \rangle, C') ,$$

where C is the discourse context before the assumption is discharged, while C' is the resulting discourse context.

Semantic-interpretation rules are obligatory in that some semantic-interpretation rule associated with a given syntactic rule must be applied to any phrase analyzed by the syntactic rule. In contrast, the application of pragmatic-discharge rules is optional, although discharging a particular assumption too early or too late may lead to a dead end in the interpretation process. Applying the same discharge rule at different points in the interpretation process for some utterance may lead to alternative interpretations, as we shall illustrate with the examples in Sections 9.6 and 9.7.

Given a sentence and its syntactic analysis, the interpretation process applies semantic-interpretation and pragmatic-discharge rules, according to their applicability conditions, to construct the derivation of a complete interpretation of the sentence. In Candide, this process resembles a syntax-directed translation system [1]. Interpretation starts at the root node of the analysis tree. For each node of the tree, the interpretation process selects an appropriate semantic-interpretation rule and calls itself recursively for each of the node's daughters. Interpretations are constructed on return from the recursion, and pragmatic-discharge rules are optionally applied in a discharge cycle that follows each application to a node of a semantic-interpretation rule.

Lexical ambiguity, multiple sem–an–tic-inter–pre–ta–tion rules for a given syntactic construction, optional application of discharge rules, and alternative ways of discharging a given assumption, are all sources of nondeterminism in the interpretation process, which need to be somehow controlled. In Candide, we adopted four simple control tactics: overall depth-first search, early discharge of assumptions, breadth-first search for alternative bindings of a discharged parameter, and bounds on assumption percolation wherever it can be shown that an assumption would not be dischargeable outside a certain syntactic domain. For lack of space, a fuller discussion of these heuristics will be conducted elsewhere [24].

9.4 The Discourse Context

Pragmatic-discharge rules need access to a discourse context that encodes information about relevant world knowledge and the discourse history. Although our framework for semantic and pragmatic interpretation can accommodate alternative representations of the discourse context, the specific discharge rules we have written and incorporated into the Candide system rely on a particular representation comprising four parts: *immediate context*, *local context*, *global context*, and a *knowledge base*.

During the analysis of a sentence, the immediate context contains detailed information about the entities referred to in that sentence; it is used primarily for resolving intrasentential anaphora. The local context generally contains detailed information about the immediately preceding sentence,[3] while the global context includes somewhat less detailed

[3]This will not be true when a "pop" of the global context has just occurred [13].

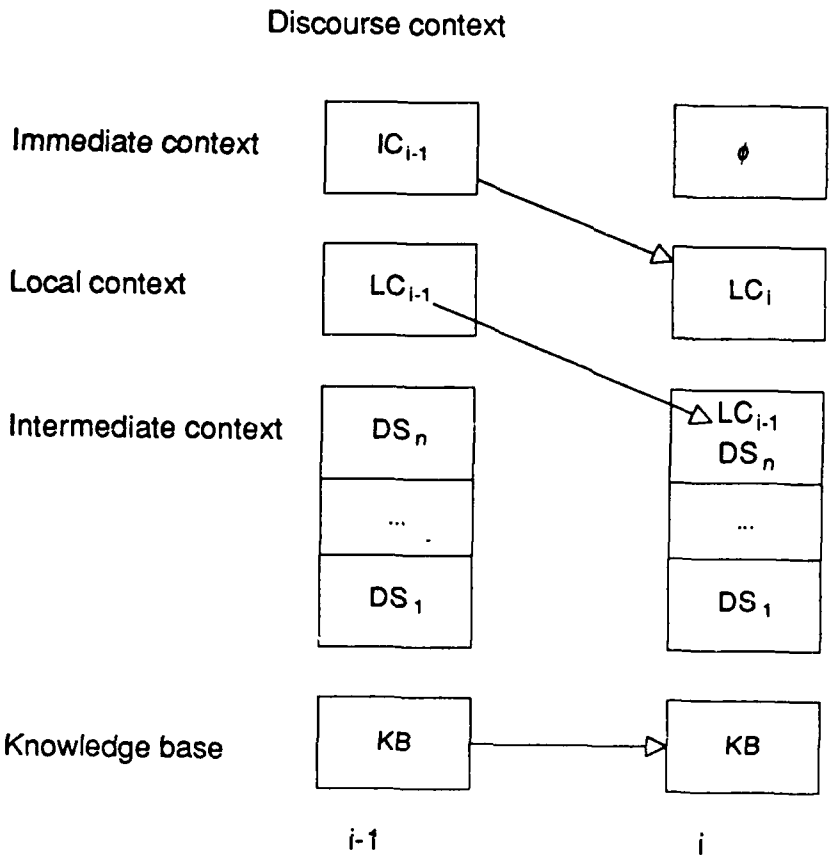


Figure 9.1: Updating the Discourse Context

information about entities referred to throughout longer stretches of the discourse. We use the local context primarily for pronoun resolution, following the theory of centering introduced by Grosz *et al.*[12]. The global context is employed primarily for the resolution of definite anaphora, and is structured as a stack to make use of the theory of focusing. Each element of the global-context stack is itself a list of entries containing information about the entities referred to in a discourse segment [13]. We refer to the top element of the global context as the *intermediate context.*

Individual discharge rules used in processing a sentence can extend the immediate context for that sentence. For instance, the rule mentioned earlier that binds a parameter as a definite reference adds to the immediate context an entry for the entity to which the parameter is bound. When the assumption in (9.1) is discharged, resulting in the interpretation in (9.2), an entry for j must be added to the immediate context. The entry will include the sort of j (*jet*) and the surface position of that phrase in the sentence (subject).

The discourse context must also be updated after each sentence has been processed. In the simplest case, the update will be quite straightforward, as illustrated in Figure 9.1: the current immediate context will become the new local context, while a subset of the information encoded in the immediate context will be also added to the intermediate context (the topmost element of the global-context stack). The immediate context will be cleared in preparation for the next sentence. For the moment, we shall assume that the knowledge base is static, although it will ultimately have to be reorganizable dynamically so as to reflect a language user's current perspective.

In fact, the update function can be rather more complex. For example, if the current utterance is recognized to be the start of a subordinate discourse segment, a new, empty element can be pushed onto the global-context stack after the local context has been merged into the previous top element. We shall discuss the discourse-context update function further elsewhere [24].

9.5 A Simple Discourse

The following simple discourse will provide our first illustration of the interpretation mechanism and, in particular, the treatment of reference and coreference:

The jet failed. (9.3)
Close the manifold.

In the subsequent sections, we shall turn to more complex examples that provide further insight into the way in which pragmatic processes can interact with one another affecting syntactic and semantic decisions.

The three semantic-interpretation rules given in Figure 9.2 are needed in the example. Recall that the interpretation process is driven by semantic-interpretation rules, which apply compositionally to phrases. Each such rule has three parts: an *applicability condition* (AC), a set of *selection functions* (SF), and a *conditional-interpretation function* (CIF). The applicability condition specifies the syntactic type of phrase to which the semantic interpretation rule applies; it is stated in terms of a predicate on trees.[4] The selection functions specify how to access the constituents of the phrase to which the rule is to be applied. Finally, the conditional-interpretation function defines the conditional interpretation of the phrase as a function of the conditional interpretations of its constituents. A conditional-interpretation function will often depend separately on the sense and assumptions of a conditional interpretation I, for which we use the notations I_S and I_A, respectively.

Figure 9.3 shows an annotated tree[5] representing the derivation of a complete interpretation of the first sentence in (9.3). Conditional interpretations of constituents of the

[4]The meanings of the predicates on trees used in this paper should be clear from their names.

[5]Our analysis trees are closer to the functional structures of lexical-functional grammar [4] than to the usual surface constituent structures. The sample analyses have been extremely simplified for expository reasons; terminal nodes, in particular, appear in the trees simply as the corresponding word, but their actual representation, as required by interpretation rule [lex], has two branches: wordstem for the actual root form of the terminal, cat for its syntactic category. Finally, tree nodes relevant to the discussion are numbered for ease of reference.

[iv-clause]:

AC: $\mathsf{intrans\text{-}verb\text{-}clause}(T)$
SF: $\mathsf{pred}(T) = V, \mathsf{arg1}(T) = A$
CIF: $[\![T]\!] = \langle [\![V]\!]_S, [\![V]\!]_A \cup [\![A]\!]_A \cup \{\mathsf{restrict}(\mathsf{arg1}, =, [\![A]\!]_S)\}\rangle$

[def-np]:

AC: $\mathsf{def\text{-}np}(T)$
SF: $\mathsf{arg1}(T) = N$
CIF: $[\![T]\!] = \langle x, [\![N]\!]_A \cup \{\mathsf{bind}(x, \mathsf{def}, [\![N]\!]_S)\}\rangle$

[lex]:

AC: $\mathsf{lex\text{-}item}(T)$
SF: $\mathsf{wordstem}(T) = W$
CIF: $[\![T]\!] = I_W$

Figure 9.2: Semantic-Interpretation Rules I

complete sentence are shown above the root nodes of the corresponding subtrees.

Semantic-interpretation rule [lex] applies to lexical subtrees (Nodes 2 and 5 in Figure 9.3[6]) associating with each wordstem W conditional interpretations I_W according to the lexicon. The lexical entries relevant to the current discussion are:

$$I_{\text{jet}} = \langle jet, \phi \rangle$$
$$I_{\text{fail}} = \langle fail(x), \{\mathsf{bind}(x, \mathsf{arg1}, device)\}\rangle$$

In the conditional interpretation of a common noun, the sense is always a sort term. The assumption set may be empty, as it is for "jet" above, but for a relational noun it will contain bind assumptions for the relation's arguments, binding parameters occurring in the sort term.

The lexical entries for verbs and the structural rules that combine a verb with its subject and complements must refer, through assumptions, to the grammatical functions that provide the arguments of the predicates that represent the senses of verbs (roughly the *governable* grammatical functions of lexical-functional grammar [4,27]). Since we are not defending any particular theory of grammar in this paper, we shall skirt a theoretical and terminological minefield by naming the grammatical functions relevant to our purposes $\mathsf{arg}i$ for $i = 1, \ldots, n$, and calling them simply "arguments." Arguments are used as edge labels in our analyses, as well as in bind and restrict assumptions, and their intended interpretation should be clear from the examples we are discussing.

The encoding of selectional restrictions is illustrated here in the conditional interpretation of the verb "fail," which is $fail(x)$, under the assumption that x must be bound as

[6] Node 4 is also lexical, but definite determiners contribute only to the interpretation of their mother noun phrase, by rule [def-np], rather than being given a separate interpretation.

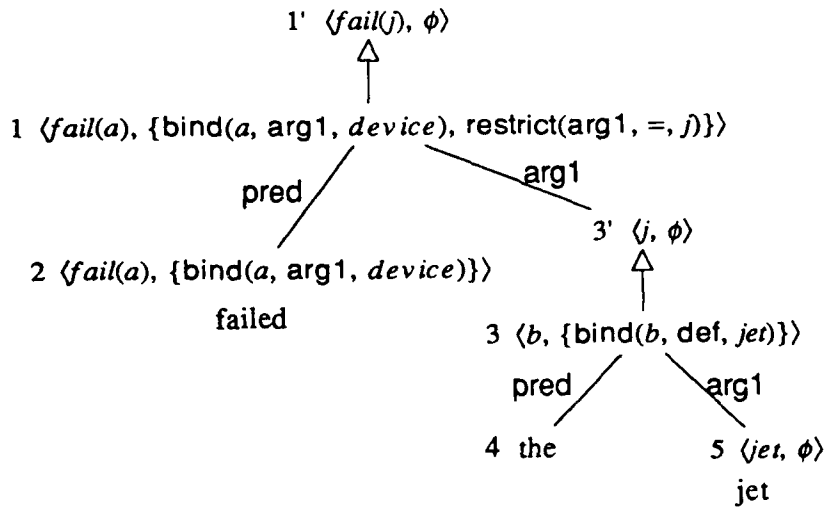


Figure 9.3: Interpretation of "The jet failed"

first argument of the verb to something of the sort *device*. This interpretation effectively encodes the information that things that fail are devices.[7]

Because the local tree rooted at Node 3 represents a definite noun phrase, rule [def-np] applies to it in a straightforward fashion, yielding the conditional interpretation

$$\langle b, \{\mathsf{bind}(b, \mathsf{def}, jet)\}\rangle \tag{9.4}$$

That is, "the jet" is interpreted as b under the assumption that b can be bound in accordance with the constraints of definite reference to an entity of sort *jet*.

As mentioned earlier, a pragmatic-discharge rule *may* be used whenever it is applicable to some conditional interpretation in context. In the current example, the rule for discharging the bind assumption is applicable to the conditional interpretation in (9.4), and it is actually used in the derivation to determine a referent for the definite noun phrase.

The process of resolving a definite reference is of course quite complex [5,11,28,29], and the rule that discharges assumptions to bind a parameter as a definite reference must reflect this complexity. For the moment, let us assume that there is only one entity of the correct sort available for definite reference (perhaps introduced in a preceding portion of the discourse): the jet identified as j. The pragmatic discharge rule can thus bind the parameter b to j, extend the immediate context accordingly, and delete the bind assumption from the list of assumptions in the current conditional interpretation. The resulting conditional interpretation of the string "the jet" is $\langle j, \phi\rangle$, shown in Figure 9.3 above Node 3'.

[7]The conditional interpretation shown above Node 2 in the figure has a new parameter, a, substituted for the variable x of the lexical entry because parameters introduced through bind assumptions in distinct applications of semantic interpretation-rules in a derivation must be themselves distinct.

Finally, consider the interpretation of the whole sentence. Rule [iv-clause] applies to the parse tree for the sentence, specifying that its sense is the sense of the predicate (**pred**) constituent, namely *fail*(a), and that its set of assumptions is the union of (i) the assumptions from its predicate constituent, (ii) the assumptions from its argument (**arg1**) constituent, and (iii) the new assumption **restrict**$(\textbf{arg1}, =, j)$, where j is the sense of the argument constituent. The restrict assumption, which arises from the sentence's syntactic form, applies to whatever parameter is to be bound as the first argument of the sense of the sentence—in this case, a, as specified by the bind assumption inherited from the predicate constituent. The restrict assumption further constrains the binding of this parameter by requiring that it be equated with the entity j.

The interpretation process is completed after the two remaining assumptions are discharged, as indicated at the top of Figure 9.3.[8] They can be discharged successfully in parallel: binding a to j is legitimate because j is a jet, and *jet* is a subsort of *device*. Before the next sentence is processed the discourse context needs to be updated, as described earlier.

The second sentence of our example is "Close the manifold"; we shall be concerned primarily with the way in which the reference resolution problem is handled. The conditional interpretation for the definite noun phrase "the manifold" is

$$\langle c, \{\textbf{bind}(c, \textbf{def}, \textit{manifold})\}\rangle \tag{9.5}$$

Discharging the bind assumption here requires the use not only of world knowledge — namely, that each jet is attached to one and only one manifold — but also of knowledge of the discourse history—namely that there is a single salient jet in context, the one identifed as j. The latter information can be derived from the discourse context, while the former must be encoded in the knowledge base. This information is sufficient to resolve the reference in the sentence under consideration: "the manifold" refers to the manifold that is attached to j. Hence the interpretation we derive from (9.5) is

$$\langle m, \phi\rangle\ , \tag{9.6}$$

where m is the unique manifold attached to jet j. For use in constraining subsequent reference, the discourse context must be updated with the information that m has the restricted sort : $\textit{manifold} \mid \lambda x.\textit{attached-to}(x, j)$, where $s|P$ is the subsort of s whose elements satisfy property P.

9.6 Quantifier Scope

We shall now turn to the kind of interactions in pragmatic processing that challenge compositional systems. In this section we shall discuss an example of quantifier scope ambiguity;

[8] In the Candide system as it currently exists, a bind assumption encoding a selectional restriction and a restrict assumption encoding the filler of an argument must be discharged as soon as the latter has been is introduced; otherwise an erroneous interpretation might be derived if the restrict assumption is mistakenly applied at a higher clause node. A better scheme would encode sufficient information in these restrict assumptions to ensure that they could apply only to the appropriate clause.

following that, we shall give an example of our analysis of donkey sentences, involving interactions between quantifier scoping and reference resolution.

The following sentence illustrates the quantifier scoping problem in its simplest form:

Every driver controls a jet. (9.7)

This sentence might be given either a wide-scope existential ($\exists\forall$) interpretation, in which all the drivers control the same jet, or a narrow-scope existential ($\forall\exists$) interpretation, in which each driver controls its own, possibly different, jet.

[tv-clause]:

AC: $\mathsf{trans\text{-}verb\text{-}clause}(T)$
SF: $\mathsf{pred}(T) = V, \mathsf{arg1}(T) = A_1, \mathsf{arg2} = A_2$
CIF: $[\![T]\!] = \langle [\![V]\!]_S,$
$[\![V]\!]_A \cup [\![A_1]\!]_A \cup [\![A_2]\!]_A \cup$
$\{\mathsf{restrict}(\mathsf{arg1}, =, [\![A_1]\!]_S),$
$\mathsf{restrict}(\mathsf{arg2}, =, [\![A_2]\!]_S)\}\rangle$

[gen-quant]:

AC: $\mathsf{gen\text{-}quant}(T)$
SF: $\mathsf{pred}(T) = Q, \mathsf{arg1}(T) = N$
CIF: $[\![T]\!] = \langle x, [\![N]\!]_A \cup \{\mathsf{bind}(x, [\![Q]\!]_S, [\![N]\!]_S)\}\rangle$

[indef-np]:

AC: $\mathsf{indef\text{-}np}(T)$
SF: $\mathsf{arg1}(T) = N$
CIF: $[\![T]\!] = \langle x, [\![N]\!]_A \cup \{\mathsf{bind}(x, \mathsf{indef}, [\![N]\!]_S)\}\rangle$

Figure 9.4: Semantic-Interpretation Rules II

Interpreting (9.7) requires additional rules of semantic interpretation, shown in Figure 9.4, and the lexical entry

$$I_{\text{control}} = \langle \mathit{controls}(x, y), \{\mathsf{bind}(x, \mathbf{arg1}, \mathit{device}), \mathsf{bind}(y, \mathbf{arg2}, \mathit{device})\}\rangle$$

Derivations of the $\exists\forall$ and the $\forall\exists$ interpretations are shown in Figures 9.5 and 9.6, respectively.

In both derivations, the general noun phrase "every driver" is interpreted at Node 2 by rule **[gen-quant]** and the indefinite noun phrase "a jet" is interpreted at Node 1 by rule **[indef-np]**. However, the two derivations differ as to where the indefinite-reference assumption is discharged. In Figure 9.5 the assumption is discharged immediately after its

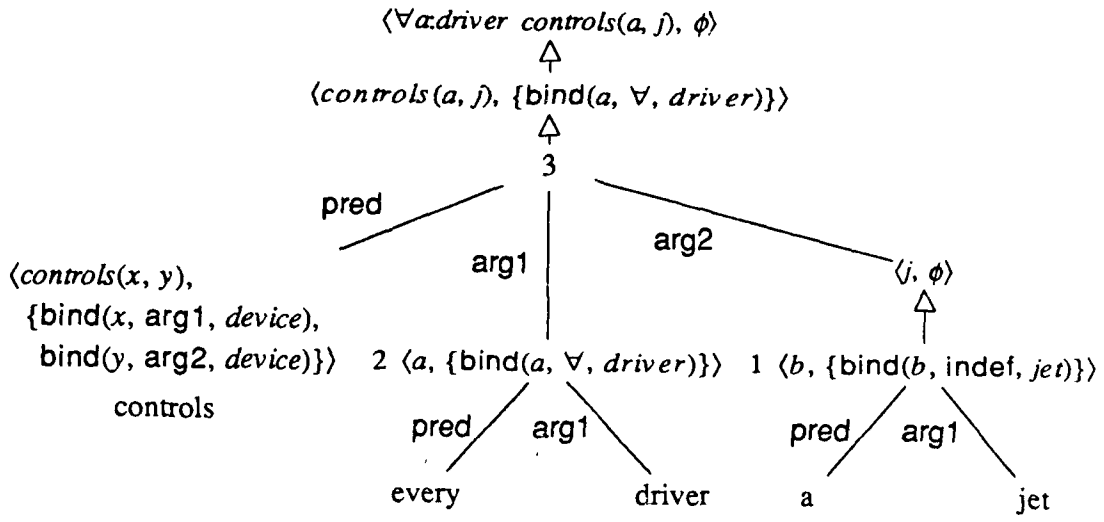


Figure 9.5: ∃∀ Interpretation

introduction. The resulting sense is a new entity j of sort *jet*. The same ∃∀ reading could also be derived by allowing the indefinite-reference assumption to percolate up to Node 3, but then discharging it before the generalized quantifier assumption. In either case, the immediate context is updated at the time of the discharge with an entry for the new entity j.

Somewhat more interesting is the derivation of the ∀∃ reading, shown in Figure 9.6. The indefinite-reference assumption is allowed to percolate to Node 3, where the generalized-quantifier assumption is discharged. This discharge applies a quantifier to its scope, but it also selects some subset of the outstanding indefinite-reference assumptions in the current conditional interpretation and discharges them, by existential quantification of the respective parameters, within the scope of the generalized quantifier. In our example, the rule converts the conditional interpretation

$$\langle controls(a, b),\ \{\mathsf{bind}(a, \forall, driver), \mathsf{bind}(b, \mathsf{indef}, jet)\}\rangle$$

into the completed interpretation

$$\langle \forall a : driver\ \exists b : jet\ \ controls(a, b), \phi\rangle$$

9.7 A Donkey Sentence

We can now discuss the more complicated interactions between assumptions occurring in donkey sentences. Our example will be the sentence:

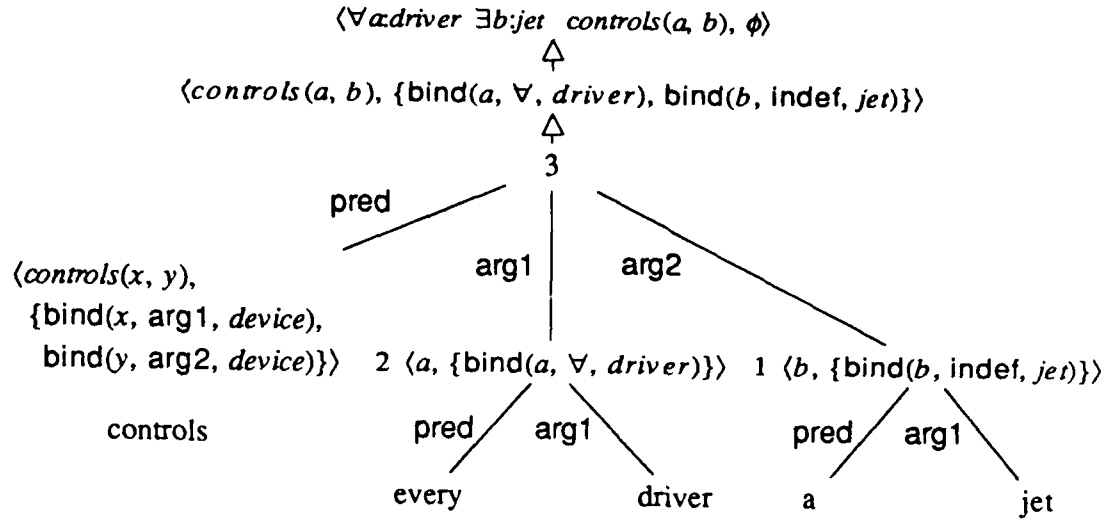


Figure 9.6: ∀∃ Interpretation

Every driver controlling a jet closes it. (9.8)

Clearly, this sentence has an interpretation in which, for every driver controlling a jet, the driver closes the jet. However, it is difficult to see how this interpretation can be derived compositionally. The well-recognized problem is that, in the intended reading, the indefinite noun phrase "a jet" has narrower scope than the determiner "every," forcing its interpretation to be part of the sort term translating the nominal "driver controlling a jet." But this means that the interpretation of the pronoun "it" will be outside the scope of the indefinite "a jet."

Our solution to the problem of interpreting donkey sentences involves two new mechanisms: *capture* rules that allow the quantifier in a general noun phrase to discharge in a particular way bind assumptions derived from singular noun phrases occurring in the general noun phrase, and a *pronoun resolution* rule that discharges a pronoun-introduced bind assumption by replacing the assumption's parameter with the parameter bound by the assumption for a possible antecedent of the pronoun.

Figure 9.7 shows a simplified derivation of an interpretation of sentence (9.8), with some of the less interesting assumptions discharged immediately after their introduction rather than being listed explicitly. Before discussing the main points of this example we need to explain our somewhat nonstandard representation of [reduced] relative clauses, as in the compound nominal "driver controlling a jet" (Node 2). A relative clause is represented as a main clause but has one of its argument positions filled by a nominal (the head noun modified by the relative clause) instead of a noun phrase. The discharge rule discussed in Section 9.5 that combines a verb argument with its filler then has two versions: one in which the filler sense is an entity, already described, and one in which the filler sense is a sort. In

the latter case, the rule produces an interpretation whose sense is the filler sort restricted by the sense of the clause. In the foregoing example, the sort-filler discharge rule is applied to the interpretation

$$\langle \mathit{controls}(x, b),$$
$$\quad \{\mathsf{bind}(x, \mathsf{arg1}, \mathit{device}),$$
$$\quad \mathsf{restrict}(\mathsf{arg1}, =, \mathit{driver}),$$
$$\quad \mathsf{bind}(b, \mathsf{indef}, \mathit{jet})\}\rangle$$

to produce the restricted sort

$$\langle \mathit{driver} | \lambda x.\mathit{controls}(x, b), \{\mathsf{bind}(b, \mathsf{indef}, \mathit{jet})\}\rangle$$

After these preliminaries, we can go on to the main point of the example. The first observation to make is that the sentence has an alternative (albeit unlikely) interpretation in which "a jet" is taken to refer to a specific jet that every driver controls. This interpretation would be derived by discharging the corresponding indefinite-reference assumption at Nodes 1 or 2 in the derivation.[9] We shall assume that this is not done, and that the indefinite-reference assumption is therefore available at Node 3.

So far bind assumptions have been given as triples of a parameter, a binding criterion (derived from a determiner), and a sort restriction for the parameter. In fact, a fourth component of *dependencies* is in general required, a set of other assumptions that the given assumption may depend on.[10] An assumption α (the *dependent* assumption) *depends* on another assumption β (the *independent* assumption) whenever the parameter for β occurs in the sort constraint of α. For the language fragment under discussion, α would be the bind assumption for a complex noun phrase and β the bind assumption for a noun phrase within a prepositional phrase or relative clause in the complex noun phrase. For correct binding of quantified parameters, semantic interpretation and discharge rules must maintain the invariant that assumptions on which a given assumption depends can occur only in its set of dependencies. Consequently, whenever a dependent assumption α is introduced any other assumption on which it depends must be moved into α's dependencies, thereby becoming inaccessible to discharge rules. If α is later discharged, the assumptions in its set of dependencies again become accessible to discharge rules. Semantic interpretation must be modified to fit this analysis. For instance, rule **[gen-quant]**, given earlier, should be instead

[gen-quant']:

AC: $\mathsf{gen\text{-}quant}(T)$
SF: $\mathsf{pred}(T) = Q, \mathsf{arg1}(T) = N$
CIF: $[\![T]\!] = \langle x, \{\mathsf{bind}(x, [\![Q]\!]_S, [\![N]\!]_S, [\![N]\!]_A)\}\rangle$

[9] A third interpretation is also possible, in which "a jet" is interpreted as a narrow-scope (nonreferential) existential, and "it" is interpreted as having an extrasentential referent. Limitations in Candide's handling of nonreferential indefinites preclude this reading, but a somewhat different rule system will generate all three readings correctly [23].

[10] In the examples so far this set has been empty and therefore omitted for the sake of clarity.

In Figure 9.7, this rule is applied at Node 3.

Capture may occur whenever a generalized-quantifier assumption with a nonempty set of dependencies D is discharged. Any indefinite assumption in D may be captured by turning it into a universal-quantification assumption and putting it into the set of assumptions for the new conditional interpretation. In our example, the indefinite assumption for "a jet" is captured in the discharge of the universal assumption for "every driver...", from Node 4 to Node 4′ in the derivation. The resulting assumption is now universal. If this assumption were discharged immediately, there would be no way of discharging the pronoun assumption as an intrasentential anaphoric reference. Instead the pronoun resolution rule is applied to discharge the pronoun assumption, causing identification of the pronoun parameter e with the jet parameter b. The resulting conditional interpretation is 4″. Finally, the remaining assumption can be discharged by quantification leading to the complete interpretation at Node 4‴.

The example shows how assumptions allow interactions between reference and quantification to be left unresolved until all the necessary information becomes available. Early discharge of the assumption for "a driver" blocks the desired interpretation for the pronoun "it"; capture makes available the attributive use of "a driver" at an appropriate point for its identification with the direct object of "close."

9.8 Related Research

Strictly compositional approaches to semantic interpretation, such as Montague grammar [19], have so far proved inadequate for dealing with interactions between meaning and context; reasons for this are noted in Section 9.1. Our approach can be thought of as a generalization of the compositional mechanism of Cooper storage [6], or of its computational analogue developed by Woods [30]. Alternative approaches that attempt to address these interactions include discourse-representation theory (DRT) [14,18] and Barwise's partial-valuation approach [2].

In DRT, the interpretation of a sentence is derived in a compositional manner from an intermediate representation called a discourse-representation structure (DRS). However, the rules that have been developed for constructing DRSs are not themselves compositional. According to the DRS-construction rules presented by Kamp [18], the DRS for a phrase is found only as a by-product of finding the DRS for the embedding discourse. In particular, DRS-construction rules apply only after the relative scope of noun phrases and anaphoric bindings have been determined. It is conceivable that our notion of conditional interpretation might be reexpressible in DRT terms, leading to a compositional system for DRS construction.

Barwise [2] uses the notion of partial valuation, that is, partial assignments of values to variables, to analyze the sorts of interactions exemplified by the donkey sentences. Similar comments apply to Webber's work [29]. In addition, none of the aforementioned accounts has been concerned with as wide a range of phenomena as is currently handled in Candide.[11]

[11]To date, we have included capabilites for processing reference and coreference (dennite and indefinite

One of the motivations for our work has been to see how Barwise's direct-interpretation approach could be turned into a two-stage one in which phrases are first "compiled" into conditional interpretations, which are then "executed" by applying pragmatic-discharge rules.

Finally, several other computational systems developed recently are concerned with interactions between context and meaning, especially Pundit [8,21] and Tacitus [17,16]. Both these systems have emphasized solutions to such difficult pragmatic problems as reference resolution. In particular, the Pundit project has made notable progress on the question of resolving missing arguments, while the Tacitus group has done the same for questions involving the determination of implicit relations. In Candide, solutions to such pragmatic problems should be encoded in the procedures that discharge assumptions; in future versions of the system the discharge procedures might be improved applying some of the techniques developed in this other work. What neither Pundit nor Tacitus has been concerned with is the question of how to build interpretations compositionally. Both systems first build partial interpretations of sentences, and then attempt to solve a collection of associated pragmatic problems. Pundit does the latter in an overly constrained way, with the result that it cannot handle systematically the sort of interactions exemplified by the donkey sentences. Tacitus, on the other hand, casts all the pragmatic problems as theorems to be proved; the result is an underconstrained control strategy. We believe that the generally compositional approach developed in Candide enables us to avoid both these extremes.

9.9 Further Work

We have developed a mechanism of semantic and pragmatic interpretation that relaxes the constraints of compositional semantics just enough to allow pragmatic information to play its necessary role in the derivation of sentence interpretations. Central to the mechanism are conditional interpetations, which allow us to separate constraints on interpretation that depend only on syntactic structure, represented by the sense component of the conditional interpretation, from those that depend on pragmatic choices, represented by the assumption component. The interpretation process is carried out by a combination of semantic-interpretation rules, which build conditional interpretations of phrases on the basis of lexical and syntactic information, and pragmatic-discharge rules, which satisfy assumptions on the basis of discourse and domain information. While the system we have implemented deals with a variety of semantic and pragmatic phenomena, of which only a few were discussed in this paper, it can only be seen as a first limited instantiation of a system architecure that requires much further work. We shall mention now a few of the directions that might be pursued in developing the architecture further.

At the most theoretical level, it is interesting to note the formal similarity of our interpretation rules to rules in "deductive" models of programming language semantics [25]. It is also interesting to consider the connection between conditional interpretations and the

noun phrases, pronouns, possessives, and proper nouns), quantifier scope, compound nominals, prepositional-phrase attachment, and certain types of underspecified relations (e.g., main-verb "have"). We shall report on these mechanisms elsewhere [24].

relational theory of meaning from situation semantics [3]. These two similarities might be fruitful in developing a semantic justification for our formal interpretation rules in terms of constraints on interpretation relations.

The applicability of discharge rules depends in many cases on the compatibility of expected and supplied sorts for relation arguments. In general, these sorts may be parameterized by assumption parameters, and some semantic interpretation problems not considered here suggest that higher-order parameterized types, instead of first-order sorts, may be needed. A suitable notion of type subsumption for such higher-order parameterized types [15] would be useful. More generally, the whole architecture would benefit from a semantically grounded treatment of parameters and parameterized objects.

Other pragmatic processes associated with discharge rules, such as those for reference resolution, also must be able to reason with parameterized objects—for example in checking the uniqueness of a dependent object relative to arbitrary parameter assignments. Ultimately, the proper treatment of singular noun phrases in context will require a closer connection between assumptions and [parameterized] fragments of the discourse context.

Acknowledgments

We would like to thank David Israel, Ray Perrault, and Stuart Shieber for their helpful discussion regarding this work.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Massachusetts, 1985.

[2] J. Barwise. Noun phrases, generalized quantifiers and anaphora. In P. Gärdenfors, editor, *Generalized Quantifiers: Linguistic and Logical Approaches*, pages 1–29, D. Reidel, Dordrecht, Netherlands, 1987.

[3] J. Barwise and J. Perry. *Situations and Attitudes.* MIT Press, Cambridge, Massachusetts, 1983.

[4] J. Bresnan and R. Kaplan. Lexical-functional grammar: a formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–281, MIT Press, Cambridge, Massachusetts, 1982.

[5] D. Carter. *Interpreting Anaphors in Natural Language Texts.* Ellis Horwood, Chichester, England, 1987.

[6] R. Cooper. *Quantification and Syntactic Theory.* D. Reidel, Dordrecht, Netherlands, 1983.

[7] S. Crain and M. Steedman. On not being led up the garden path: the use of context by the psychological syntax processor. In D. Dowty, L. Karttunen, and A. Zwicky, editors, *Natural Language Parsing: Psychological, Computational, and Theoretical Perspectives*, pages 320–358, Cambridge University Press, Cambridge, England, 1985.

[8] D. A. Dahl, M. S. Palmer, and R. J. Passonneau. Nominalizations in Pundit. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 131–139, Stanford, California, 1987.

[9] P. Geach. *Reference and Generality.* Cornell University Press, Ithaca, New York, 1962.

[10] M. P. Georgeff and A. L. Lansky. Procedural knowledge. *Proceedings of the IEEE, Special Issue on Knowledge Representation*, 1383–1398, 1986.

[11] B. J. Grosz. *The Representation and Use of Focus in Dialogue Understanding.* Technical Report 151, SRI International, Menlo Park, Ca., 1977.

[12] B. J. Grosz, A. K. Joshi, and S. Weinstein. Providing a unified account of definite noun phrases in discourse. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, pages 44–50, Cambridge, Massachusetts, 1983.

[13] B. J. Grosz and C. L. Sidner. Attention, intentions, and the structure of discourse. *Computational Linguistics*, 12(3):175–204, 1986.

[14] F. Guenthner, H. Lehmann, and W. Schonfeld. A theory for the representation of knowledge. *IBM Journal of Research and Development*, 30(1):39–56, January 1986.

[15] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proceedings of the Second Symposium on Logic in Computer Science*, Cornell University, IEEE, Ithaca, New York, 1987.

[16] J. R. Hobbs. Interpretation as abduction. In *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics*, Buffalo, New York, 1988.

[17] J. R. Hobbs. Overview of the TACITUS project. *Computational Linguistics*, 12(3), 1986.

[18] H. Kamp. A theory of truth and semantic interpretation. In J. A. G. Groenendijk, T. M. V. Janssen, and M. B. J. Stokhof, editors, *Formal Methods in the Study of Language*, pages 277–322, Mathematisch Centrum, Amsterdam, Netherlands, 1981.

[19] R. Montague. The proper treatment of quantification in ordinary English. In R. H. Thomason, editor, *Formal Philosphy*, Yale University Press, 1973.

[20] R. C. Moore. Problems in logical form. In *Proceedings of the 19th Meeting of the Association for Computational Linguistics*, pages 117–124, Stanford, California, 1981.

[21] M. S. Palmer, D. A. Dahl, R. J. Schiffman, L. Hirschman, M. Linebarger, and J. Dowding. Recovering implicit information. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 10–19, New York, 1986.

[22] F. C. Pereira. *Logic for Natural Language Analysis.* Technical Report 275, SRI International, Menlo Park, California, 1983.

[23] F. C. Pereira. Towards a deductive theory of sentence interpretation. Unpublished manuscript.

[24] F. C. Pereira and M. E. Pollack. A compositional, declarative system for semantic and pragmatic interpretation. In preparation.

[25] G. D. Plotkin. *A Structural Approach to Operational Semantics.* Lecture notes DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981.

[26] L. K. Schubert and F. J. Pelletier. From English to logic: context-free computation of 'conventional' logical translation. *Computational Linguistics*, 8(1):26–44, 1982.

[27] P. Sells. *Lectures on Contemporary Syntactic Theories.* Volume 3 of *CSLI Lecture Notes*, Center for the Study of Language and Information, Stanford University, Stanford, California, 1985. Distributed by University of Chicago Press.

[28] C. L. Sidner. Focusing in the comprehension of definite anaphora. In *Computational Models of Discourse*, MIT Press, Cambridge, Massachusetts, 1983.

[29] B. L. Webber. So what can we talk about now? In *Computational Models of Discourse*, MIT Press, Cambrige, Ma., 1983.

[30] W. A. Woods. Semantics and quantification in natural language question answering. In M. Yovits, editor, *Advances in Computers, Vol. 17*, pages 2–64, Academic Press, New York, New York, 1978.

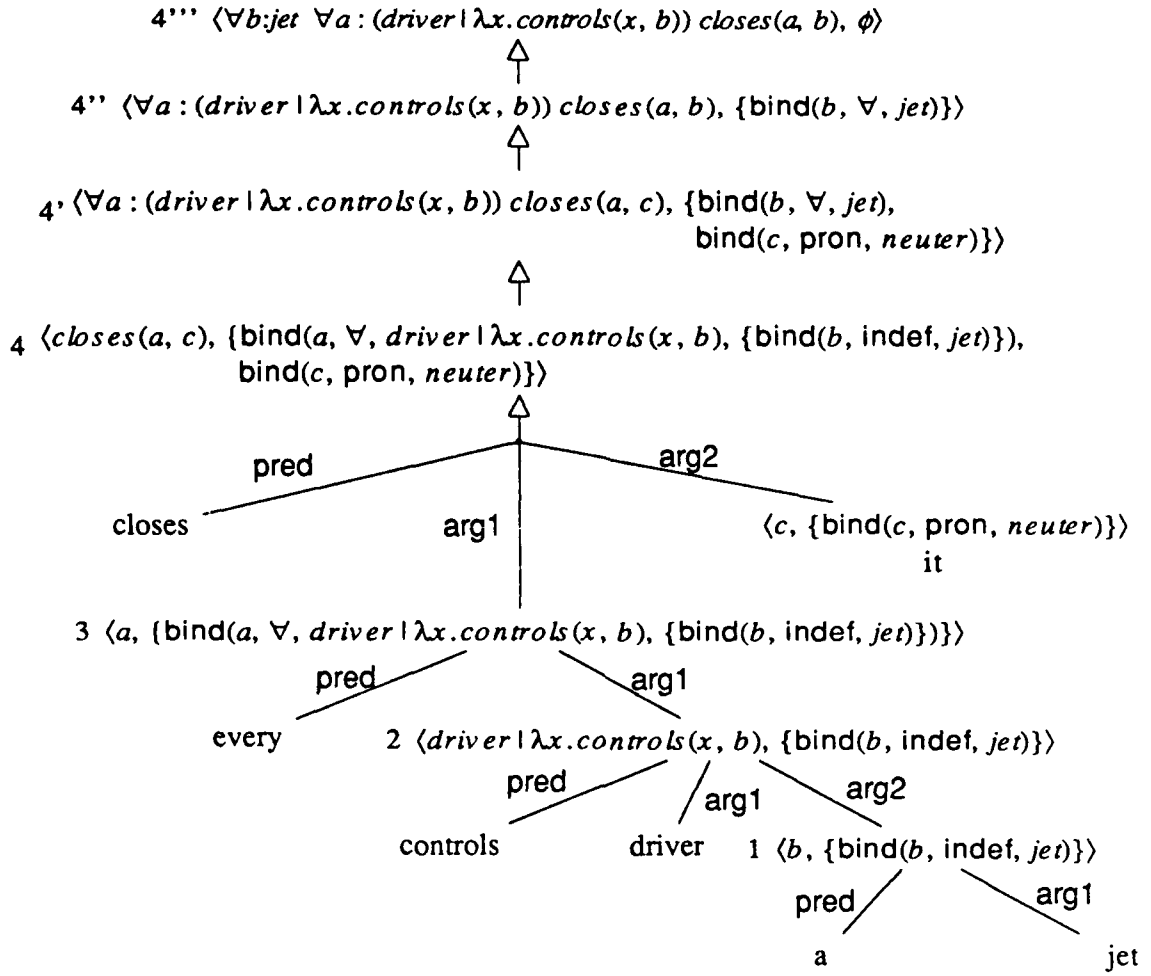


Figure 9.7: Interpretation of a Donkey Sentence

Appendix A

CANDIDE User's Manual

This appendix was written by Barney Pell.

A.1 Introduction

This manual is intended to provide a hands-on introduction to using the CANDIDE system for the creation of procedural nets which serve as input to the Procedural Reasoning System (PRS). Candide is incorporated in the GRASPER II system. This manual does not assume familiarity with GRASPER, but for advanced editing techniques the reader is referred to the GRASPER II REFERENCE MANUAL.

A.2 Creating a Procedural Net Using CANDIDE

A.2.1 Loading the Candide system

The first step in creating a procedural net is to load the Candide system. For details on loading the system, see Section A.3.

A.2.2 Using the Menus

NOTE: This section consists in a summary of "APPENDIX C: Architecture of the Graphic Interface" from the GRASPER II REFERENCE MANUAL.

The graphic interface has two main types of menus, which appear on the left side of the screen. The upper menu, the "noun menu," is used to designate which type of entity is to be manipulated. Selecting one of these items will expose the associated "verb menu" in the lower menu pane. The verbs indicate the operations that can be performed upon the entity designated by the chosen noun. Selecting a menu item is accomplished by clicking the left mouse button while the cursor is over that item. Any specific information about

the particular operation will be displayed in the mouse documentation line, at the bottom of the screen.

Throughout the system the heirarchical nature of these menus is observed. The noun menu is at the root of the heirarchy, followed by one of the verb menus, and finally the action itself. The user can pop up a level in this heirarchy at any time by clicking the right mouse button, thereby terminating the ongoing operation. Use of the right button is encouraged as it can eliminate a lot of excessive mouse movement.

A.2.3 GRAPH Operations

A *graph* is a collection of *spaces*, where each space contains one procedural net. These collections can be saved and retrieved from storage. Thus the next step in creating our procedural net (once we have loaded the Candide system and entered GRASPER II through PRS) is to select the graph it will be stored in. First we need to enter the file-name with which this graph is to be associated. To do this, click-left on the *GRAPH* option from the menu. The menu which now appears is the list of possible operations on graphs. If we want to create a new graph or edit one already loaded, we can select the *SELECT* option, again by clicking left on it. Now click-left on the file-name of the graph to be edited, or click-left on *NEW GRAPH* and enter the file-name under which this graph will be saved. Alternatively, we may wish to edit a graph which has not been loaded yet. In this case, we can select the *INPUT* option, and enter the appropriate file-name.

When we are finished creating our graph, we will use the *OUTPUT* option from this menu to save it to a file for later use.

A.2.4 SPACE Operations

We are now ready to create a space that represents a PRS procedural net. To do this we enter the *SPACE* menu by clicking the right mouse button, and then clicking-left on SPACE. Since we are creating a new procedural net, we choose the *CREATE* option, and then enter the name to associate with this space.

Now we are ready to enter the "Invocation Part" of the net, which specifies the facts or goals that must be true for this net to be applied. (For more information about procedural net's, see the literature on PRS). We enter the invocation part by selecting the *INVOCATION* option from the *SPACE* menu. Now we enter, in English, all of the invocation data. For example, if we want this space to be invoked whenever the RCS jet warning light is on, we respond to the prompt as follows:

Enter Invocation Data: *the rcs jet warning light is on* ⟨*RETURN*⟩

We proceed in this way for each item of data, and then enter ⟨RETURN⟩ when done. As each item is entered, it is processed by the CANDIDE system. When all the invocation data has been entered, the resulting logical forms are displayed in the upper left-hand part

of the space. Thus, after naming our space "jet-fail-on", and entering the invocation data shown above, the graphic window looks like Figure A.1.

If we wanted to declare the effects of this procedural net explicitly, we could enter them using the *ACTION* option, which operates the same way as the *INVOCATION* option.

If we want to display the actual English sentences that we entered, we could select the *FORMAT* option, and choose the format in which to display the data (English or Logic, chosen by clicking middle or left, respectively).

Finally, we can use the *SELECT* option to work on different spaces in our current graph.

A.2.5 NODE-EDGE Operations

Once we have entered the invocation and/or action parts, we are ready to create the body of the procedural net. This is done through operations on the *NODE-EDGE* menu.

We create the net using the *CREATE* option from the NODE-EDGE menu. This works as follows:

- A NODE is created by pointing the mouse at a location, and clicking left. Nodes are automatically numbered, and the first node created is the START node.
- An EDGE is created by clicking left on the PARENT node, and then on the CHILD node, and then entering in English the sentence which labels the edge to be created.
- Nodes which are at the bottom of the hierarchy (leaf nodes) should be converted to FINISH NODES, using the *FINISH-NODE* operation.

A.2.6 An Example

Here is a step-by-step example of the creation of a small procedural net.

Starting from Figure A.1, we select the Node-Edge menu, and then the Create option. we now create a node below the Invocation-Part. This node is labeled START because it is the first node we have created in this space.

Below it, we create a second node, and then create an edge from the first to the second. As a label for this edge, we enter:

is a jet faulty

This causes two branches to be created automatically: One for the case in which there is a faulty jet, and the other for that in which there is not a faulty jet. If no jet is faulty, then this procedure has no effect, so we change the node at the bottom of that path to a FINISH node, indicating that nothing more need be done in this case.

However, if there is a faulty jet, then the next step in this procedure should be to close the manifold attached to that jet. Thus, we create a node below the bottom node in the faulty-jet branch, and then an edge between them, which we label:

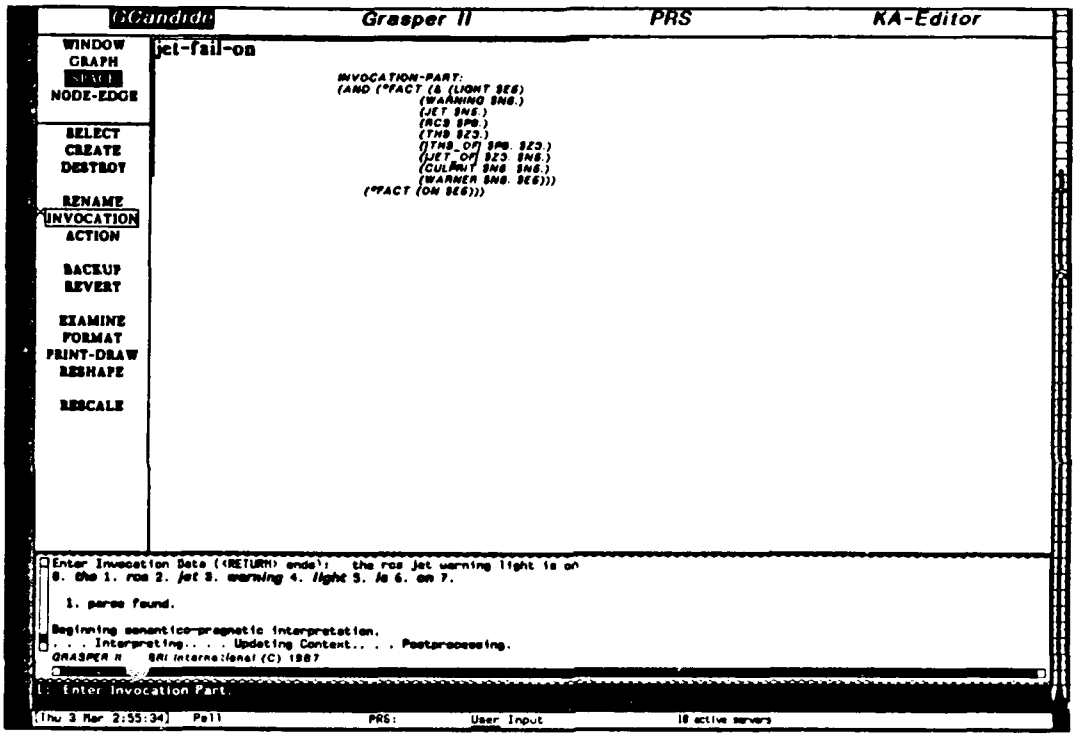


Figure A.1: Partial Procedural Net

close its manifold

Finally, we convert this last node into a FINISH node, since closing the manifold should solve the problem in my scenario. Now that all possible paths are closed off with FINISH nodes, this space is complete. We now have the space pictured in Figure A.2.

This small example illustrates one way in which Candide allows us to make use of natural dialogues in describing procedural networks. After referring to a jet in labelling the first edge, we were able to use the pronoun "it" to make subsequent reference to the same jet. That is, we were able to label the second edge with the statement "close its manifold". Candide maintains a record of the dialogue, called the discourse context. This context flows down the graph. That is, edges lower in the graph can refer to entities introduced by sentences which label edges above them. All edges can refer to the entities introduced in the sentences in the INVOCATION-PART.

When edges are labeled with interrogative edges, CANDIDE automatically creates two branches in the net, one in which the answer is "YES" (by convention, the right branch), and one in which the answer is "NO". It is important to note that the resulting discourse contexts can be different in these two cases. In the example above, the entity originally described as "a jet" is available for reference in the branch of the net corresponding to the positive answer to the question. Thus the statement "close the manifold" is properly interpreted on an edge that emanates from this branch. However, this statement would not be valid on the branch emanating from the negative answer, since no new jet was introduced into the context (the answer was effectively "No, there is no faulty jet"). Thus we see that different branches can carry different contexts, and sentences are interpreted within the context of the particular branch in which they are entered. Entities introduced by a sentence entered on one branch of a graph cannot be referred to by those on different branches.

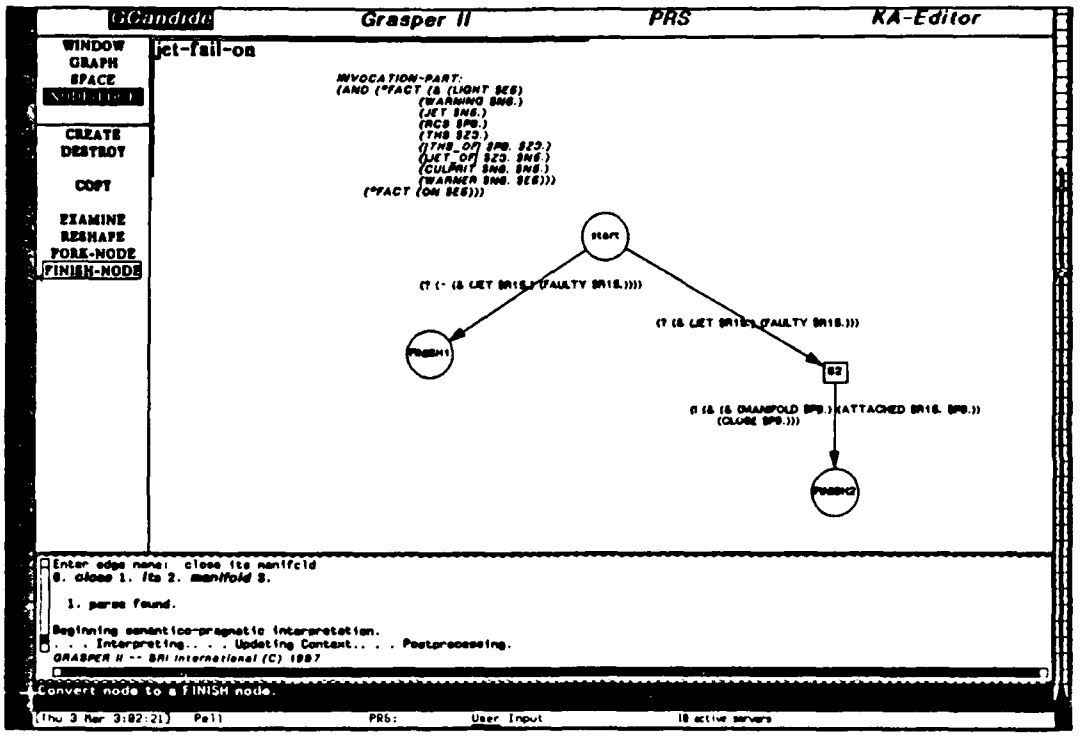


Figure A.2: Completed Procedural Net

A.3 How to load the CANDIDE system

The files used in the CANDIDE system are stored on BISHOP, a Symbolics LISP Machine in the Artificial Intelligence Center at SRI International in Menlo Park. Although the CANDIDE system is built upon several different systems, all of these systems have been assembled into one system (stored on BISHOP), so they need not be loaded individually.

For the names and locations of the individual systems and files used in this system, see Section A.4.

The rest of this section is the step-by-step procedure by which we reset the state of the machine, load the Candide system, initialize the subsystems, load the relevant grammar and database, create a graph of procedural nets, and finally run the PRS system on the graphs we have created.

A.3.1 Loading the CANDIDE SYSTEM

To load the entire Candide System, we enter the following command from the LISP Listener:

Load System Candide

A.3.2 Initializing the subsystems

We are now in a state in which all of the subsystems have been loaded (with the world), but are not yet initialized.

Initializing PATR

To initialize PATR, we enter the following function from the LISP Listener:

(zl-user:klaus-start)

This will create the windows used by the PATR system. Also, the key-sequence ⟨*SELECT K*⟩ now allows us to select the PATR system from any environment.

Once we are in PATR, we use the *PROFILE* option to set up our grammar file and to use CANDIDE as the logical form processor. Thus, we modify the following to lines in our profile to read as follows:

Name of Default Grammar System: *Candide:Candide; April87demo.patr*
Logical Form Processor: *Candide*

Now, we load this grammar by selecting the command *LOAD* from the PATR menu.

For further information on the use of PATR, see the PATR literature.

Initializing PROLOG

To initialize PROLOG, we enter ⟨*SELECT SYMBOL-SHIFT-P*⟩. Then, we respond to the PROLOG prompt as follows:

?- *lp.* ⟨*RETURN*⟩

NOTE: This last entry is necessary to overcome a bug in Symbolics PROLOG.

Initializing PRS

To initialize PRS, we enter ⟨*SELECT A*⟩. We will now be in the PRS window, and we can get to the PRS *MAIN MENU* by clicking left.

Now we load our database(s) into PRS by selecting the *LOAD* option from the main menu. This will reveal a new menu of operations which allow us to load new databases and processes (procedural nets) into PRS.

To load our first database, we select *INITIALIZE DATABASE*, and then enter the filename of the first database we want to load. For example, to use the database for the Reaction Control System of the Space Shuttle, we enter:

Candide:Candide;RCS.static ⟨*RETURN*⟩

If we want to load any more databases, we can use the *APPEND DATABASE* option.

Now we must load our "meta-procedures" graph, which describes the method by which PRS is to interpret the procedural nets we create. We do this by selecting *LOAD* from the PRS main menu, and then selecting *INITIALIZE PROCESSES*. We then indicate that we want to load our graph from a file, and then enter the filename of our graph as follows:

Candide:Candide;meta-kas.graph

Initializing GRASPER

To enter GRASPER from PRS, we select the *GRASPER* option from the PRS *MAIN MENU*. The *first time* that we enter grasper, we enter ⟨*SELECT G*⟩ to initialize GRASPER and make the *GCandide* option appear at left column of the applications menu (at the top of the screen). We then select this option (by clicking left on it), and can begin creating a procedural net as described in the body of this manual.

When we have finished creating our net, we return to PRS by clicking on the *PRS* application. We then load this graph into PRS by selecting *LOAD* from the PRS main menu, and then selecting *APPEND PROCESSES* (since we have already loaded our default procedural net, the "meta-kas"). Since we have just created a graph, we can load it directly from GRASPER, and need not enter the filename of our graph. However, it is important to

note that PRS will only load the graph which is currently selected in GRASPER. If we are editing multiple graphs, we can ensure this by using the *SELECT* option from the *GRAPH MENU* (in GRASPER) to select the graph we want to load.

For further information on PRS, see the PRS literature.

A.4 Components of the CANDIDE system

The Candide system consists of PRS, GRASPER, PATR, the LOCAL-PRAGMATICS system, and the INTERFACE MODULE.

The home directory for Candide system files is:

Candide:Candide;

This section briefly describes the function and location (when necessary) of the components of the system.

A.4.1 PRS

PRS is the Procedural Reasoning System, which reasons about procedural nets. These nets are directed graphs in which the edges are labelled in a language like first-order logic. The graphs are of the kind created using the GRASPER II system.

For more information about PRS, consult the PRS literature.

A.4.2 GRASPER II

GRASPER II is a programming language extension that supports graph processing. All of *the graphic routines used in the CANDIDE system implemented in this system.*

For more information about GRASPER II, see the GRASPER II Reference Manual.

A.4.3 PATR

PATR is a unification-based grammar formalism, which takes English sentences and constructs a feature-structure reflecting the syntactic construction of the sentence entered. The CANDIDE system uses the Zeta-LISP implementation of PATR running on the SYMBOLICS lisp machine.

For more information about PATR, see the PATR literature.

A.4.4 CANDIDE-SPI

These files, stored in the CANDIDE system directory, make up CANDIDE-SPI, the component of CANDIDE that performs semantic and pragmatic processing. Each includes *documentation that describes it particular content:*
ACCESS, ANAPHORA, ASSMS, AUX, AUXLP, CHOOSE, DICT, DISCHARGE, DISPLAY, INTERPRET, KB, OPS, RESTRICT, ROLES, SCENARIOS, SCOPE, SETOF, SPECIALS, TESTS, TOP, TYPES, UPDATE

A.4.5 INTERFACE MODULE

These files allow provide for communication between the various components, and form the user-interface to the entire system. All of these files are stored in the home directory for the Candide system, described above.

ASSIM Allows the LOCAL PRAGMATICS system to use the feature structures created by PATR.

GCANDIDE The main program that allows for the creation of procedural nets using natural language. As each sentence is entered, it is parsed (by PATR), interpreted (by LOCAL PRAGMATICS), and transformed into PRS notation (by TRANSFORM).

TRANSFORM Transforms the logical representation scheme used by the LOCAL PRAGMATICS system into the logic used by PRS.

FOLD-STRING Allows a long and complex structure to be broken into logical pieces and displayed as multiple lines in GRASPER.

A.4.6 MISCELLANEOUS FILES

These files are data files used by the main components of this system. All of these files reside on the CANDIDE system directory.

PATR Grammar files

These files are used by PATR for such information as grammar and morphology:
APRIL87DEMO.defs, APRIL87DEMO.doc, APRIL87DEMO.gram, APRIL87DEMO.lex, APRIL87DEMO.morph, APRIL87DEMO.patr

PRS Database and Graph files

These files are used by PRS to represent the external world and to provide primitive functions used in the Candide system.

RCS.STATIC The static database representing the Space Shuttle scenario to PRS.'

DEFAULT-PROCESSES.graph The primitive processes used by the Candide system.

META-KAS.graph The complex processes used by the Candide system, such as the Negate-as-failure procedure.

JET-FAIL-ON.graph The demo graph for the Space Shuttle scenario, created using the Candide system.

Appendix B

User Manual for the PATR-II Experimental System

This appendix was written by Stuart Shieber.

B.1 Structure of the User Interface

The user interface to the PATR-II Experimental System is built around a window with five panes on the Symbolics 3600 screen. We first describe the five panes at a high level, then discuss the particular options available in the menu panes and the interaction modes possible in the interaction panes.

- *Header pane.* A masthead with the name of the system. In the current case, this is "PATR-II Experimental System"; in earlier versions incorporated into the KLAUS question-answering system, the name was "KLAUS". Nothing of interest happens here.
- *Right interaction pane.* Below the header pane, the right half of the screen constitutes the right interaction pane. Natural language interaction with the system occurs here. The user is prompted with "PATR>". He can type sentences to the prompt causing them to be parsed. In addition, typing s-expressions to the prompt causes them to be eval-ed (useful for debugging the system).
- *Center menu pane.* Running down the center of the screen is a pane with several menu items. These items, invoked by clicking on them, allow for loading, clearing, and editing grammars, configuring the system, etc. If further interaction is needed to process the item, e.g., asking for a system name to load, this interaction goes on in a temporary window (see Section B.1).
- *Left interaction pane.* This pane is actually composed of several (currently seven) completely overlapping windows organized in a stack; only the top window in the stack is displayed at any given time. They are used for displaying information about the

chart, edges in the chart, grammar rules, templates, etc. For information on exposing a different window and changing the contents of a window, see Section B.2.2.

- *Left menu pane.* Above the left interaction pane is a small menu with options that control the contents of the several left interaction pane windows, and allow for exposing and deexposing them, and printing various types of information in them. See Section B.2.2 for more details.

In addition to the menus and the textual interaction in the right interaction pane, the user can interact with the system in two additional ways. First, the user can click on mouse-sensitive icons representing grammar rules, lexical entries, etc. In fact this is probably the most common form of interaction with the system. Icons can be identified in that when the mouse cursor is over an icon, a box appears surrounding the icon. Typically, clicking left on such an icon (which can appear in either the left or right interaction panes) causes detailed information about the item to be displayed in a window in the left interaction pane. Clicking right pops up a menu of actions that can be performed, usually including editing the source text that engendered the item. See Section B.2.3 for further details.

Second, interaction of a temporary or short-lived sort often occurs through pop-up windows and menus rather than the normal left interaction pane or command menus. These temporary windows provide for the appropriate user interaction and then disappear once their purpose is served.

B.2 menu options

All menu options in the two menu windows are mouse-sensitive. When the mouse cursor is placed over an item, the cursor shape changes from an arrow to a small x and a box surrounds the option text. Documentation then appears in the "mouse line," a black line at the bottom of the screen providing a brief description of what clicking on this item will do.

B.2.1 Center menu options

Load The "load" option is used for loading[1] files with text notating PATR-II grammars, lexicons, and so forth. A concept of "current system" is maintained by PATR-II. Clicking left on the option causes the current system file to be loaded. (The file name is built by adding the extension ".patr" to the name of the current system.) If there is no current system, the user is prompted for a system name, which is completed to a system file by adding directory information (usually the user's directory) and extension (usually ".patr") if these are unspecified. These defaults are presented to the user in the prompt so that they can be taken into account when specifying the system name.

[1]The terms "loading" and "compiling" are used interchangeably in this manual when applied to PATR-II files.

Alternatively, clicking middle on the option causes PATR-II to prompt the user for a new system name to become the current system, then loads the current system file (completing it as above).

Clicking right on the option causes PATR-II to prompt the user for a system name, completes it to a file name and loads that file. The current system name is left unchanged.

When loading a file, PATR-II prints the message "Loading <file-name>." Depending on the current configuration of the system (see Section B.2.1), it also prints messages every time an object is loaded (e.g., "Loaded word **uther**.") and/or prints every token in the file as it is read. All of the above interaction goes on in a temporary window.

For information about the format of PATR-II grammar files, see Section 2.3.1.

Clear This option clears all information about the grammar and lexicon that was loaded, putting the PATR-II system in roughly a nascent state. All the tables of grammar information, and so forth, are reset. A notification of completion is printed in a temporary window.

Edit Sends the user to the ZMACS editor window. This is not the usual way of getting to the editor. Usually, the user clicks the "edit" option on a mouse-sensitive icon to edit the source for that icon. (See Section B.3.1.)

Tables Prints out various tables of grammar rule information in a temporary window, which table being dependent on which mouse button was clicked.

Left: Table giving, for each nonterminal in the grammar, which nonterminals can appear along a left branch in a parse tree *under* the nonterminal, the so-called *first* relation.

Middle: Table giving, for each nonterminal in the gramar, which nonterminals can appear along a left branch in a parse tree *above* the nonterminal, the so-called *first-inverse* relation.

Right: Table giving, for each nonterminal, which grammar rules have the nonterminal as left corner (i.e., first nonterminal on the right-hand side of the phrase-structure portion of the rule)

These tables are used by the parser during parsing and are made available to the user primarily for debugging the parser, not for debugging grammars.

Profile The "profile" option is used for dynamic reconfiguration of the PATR-II system. It pops up a menu allowing various options to be set or altered. The initial values for all these options are set by loading a patr.init file from the user's directory (or to default values if no such file exists). After changing the various options, the user clicks "Exit" to actually make the changes, or "Save" to make the changes and update (or create) the patr.init file, or "Reload" to revert to the old values. The options are:

- *User name.* The user's name.

- *Enable top-down filtering during parsing.* Value can be "Yes" or "No". Default is "Yes". Self-explanatory. See discussion of parsing algorithm in Section 2.3.3.
- *Default system name.* Self-explanatory. See Section B.2.1 for information about system names.
- *Trace parsing of grammar files.* Value can be "Yes" or "No". Default is "No". If "Yes", then during loading of files, every token processed is echoed to the right interaction pane. Useful for debugging the loading package and meta-parser.
- *Trace loading of grammar files.* Value can be "Yes" or "No". Default is "No". If "Yes", then during loading of files, for every rule processed an appropriate message is printed to the right interaction pane including a mouse-sensitive icon for the object loaded, e.g., "Loaded word **stormed**" where **stormed** is a mouse-sensitive icon that can be clicked on to display or edit the word.
- *Trace building of active edges.* Value can be "Yes", "No", or "All". Default is "No". If "Yes", active edges using rules that are marked as being traced have icons echoed to the right interaction pane as they are built. If "All", active edges using all rules (i.e., all active edges) are so echoed, regardless of whether they are marked as being traced. (See Section B.3.2.)
- *Trace building of passive edges.* Same, but for passive edges. Probably more useful. (See Section B.3.2.)
- *Trace hypothesis and failed edges.* Value can be "Yes" or "No". Default is "No". Keeps track of edges that are normally not added to the chart, i.e., edges that failed during unification or edges with no constituents found (corresponding to dotted rules with the dot all the way to the left). When information about a vertex is printed, icons for these edges are also printed so that the user can look at them, see where the unifications failed, etc.
- *Trace addition of new allowed nonterminals.* Value can be "Yes" or "No". Default is "No". If "Yes", then whenever the parser decides constituents under a particular nonterminal can begin at a particular vertex, this information is reported to the user in the right interaction pane. Useful for parser debugging. See Section 2.3.3.
- *Trace failing unifications during edge-building.* Value can be "Yes" or "No". Default is "No". If "Yes", then whenever an edge fails due to unification failure, an icon of that edge is echoed to the right interaction pane. Useful for grammar debugging.
- *Call the prover after parsing.* In versions connected to the CG5 theorem prover, this option controls whether a logical form expression is extracted from the parse for each sentence, converted to first-order form and sent to the theorem-prover to do question-answering.
- *Allow epsilon rule processing.* Changes operation of loading and parsing so that grammars with epsilon rules (rules with empty right-hand sides) are processed correctly. This slows down the parser somewhat.
- *Allow filler-gap processing.* Changes operation of loading so that every grammar rule is automatically augmented to include unifications to perform passing of gap

information for doing long-distance dependencies. An esoteric feature, not to be worried about.

WFFs In versions incorporating the CG5 theorem prover, lists the current contents of the theorem-prover database.

Reset In versions incorporating the CG5 theorem prover, resets the theorem-prover database.

Window A toggle, changes the configuration of the interface so that the left interaction pane is removed, the center menu is moved to the far left and the right interaction pane is expanded to fill the rest of the screen, roughly doubling in size. This configuration is intended for use when doing theorem-prover debugging rather than grammar debugging.

B.2.2 Left Menu Options

The left menu controls the left interaction pane and its several overlapping windows (which we will call the *display windows*). The display windows are arranged in a stack (actually, a ring buffer) with the topmost window being exposed to view. Whenever information is to be displayed in the left interaction pane, the display window at the bottom of the stack is pulled to the top of the stack and the information is displayed there, all other elements of the stack are shifted back. Thus, the criterion used for choosing a free display window is basically the LRU (least recently used) method.

The stack of windows can be accessed directly via a directory of the display windows. Through this technique, any of the elements of the stack can be moved to the top and consequently exposed. This is described below in more detail under the "Directory" command.

Rotate forward: Shifts the stack forward (making the second element of the stack the first and displaying it) and moves the front element to the rear.

Rotate back: The inverse of "Rotate forward"; shifts the stack back (making the first element of the stack the second and deexposing it) and moves the back element to the front consequently displaying it.

Swap: Swaps the front two elements of the stack, thereby exposing the second element of the stack. "Swap" is its own inverse. Useful for comparing the contents of two windows.

Rules: Finds a free window (using LRU), moves it to the front of the stack (thereby exposing it) and displays a list of icons for the rules in the loaded grammar. These icons can then be clicked on to display the rules, trace them, etc. (see Section B.2.3).

Chart: Finds a free window (using LRU), moves it to the front of the stack (thereby exposing it) and displays the chart for the last parsed sentence in the window. Elements of the chart (the vertices and words) are mouse-sensitive icons (see Section B.2.3).

Templates: Finds a free window (using LRU), moves it to the front of the stack (thereby exposing it) and displays a list of icons for the templates in the loaded grammar. These icons can then be clicked on to display the templates, edit them, etc. (see Section B.2.3).

Lexical rules: Finds a free window (using LRU), moves it to the front of the stack (thereby exposing it) and displays a list of icons for the lexical rules in the loaded grammar. These icons can then be clicked on to display the lexical rules, edit them, etc. (see Section B.2.3).

Show: Prompts the user in a temporary window for a type of object to be displayed (rule, word, template, etc.) and identifier for the object (name of template, or word, unique identifying field of rule, etc.). Finds a free window (using LRU), moves it to the front of the stack (thereby exposing it) and displays the object thereby specified. These icons can then be clicked on to display the templates, edit them, etc. (see Section B.2.3). Moving the mouse away from the temporary menu aborts the command.

Directory: Prompts the user in a temporary window with a menu of the display windows ordered by their position in the stack, the front (exposed) window at the top, the least recently used at the bottom. Each element of the menu is either the string "(initially empty)" if nothing has ever been displayed in the window, "(empty)" if the window has been cleared (see the "Right" option below), or a string identifying the contents of the window. The user can click on any of the elements to perform various functions depending on which button was clicked.

Left: Move this window to the top of the stack, thereby exposing it.

Middle: Move this window to the bottom of the stack, deexposing it if necessary, thus making it available to be used as a free window next time a window is needed.

Right: Same as middle but clears the window as well as moving it to the bottom of the stack.

B.2.3 Mouse-sensitive icon options

Much of the interaction with the system is through mouse-sensitive icons associated with words, rules, edges in the chart, vertices in the chart, etc. Clicking left on an icon causes it to be displayed in a window in the left interaction pane; the window used for this purpose is the one at the bottom of the stack as usual. Clicking right on an icon pops up a menu of options one of which, the "Display" option does exactly the same thing that clicking left on the item does. The other options are discussed below.

Display: Same as clicking left on the icon. Finds a free window (using LRU), moves it to the front of the stack (thereby exposing it) and displays the object associated with the icon in the window. The information actually displayed depends on the type of object displayed.

Vertex: Displays iconic (mouse-sensitive) list of all passive and active edges coming into this vertex, i.e., with this as their final vertex. If tracing of hypothesis and failed edges is enabled (see Sections B.2.1 and B.3.2) then a list if hypothesis and failed edges is also displayed.

Edge: Displays information about the edge including: the start and end vertices, the dotted rule associated with the edge, the grammar rule used, the terminals covered by the edge, the child edges out of which this edge was formed, the DAG associated with the edge, etc.

Rule: Displays information about the rule including: the context-free part, the unique identifying field, whether the rule is currently being traced, the unifications associated with the rule, etc.

Template: Displays information about the template including: the unifications associated with the template.

Lexical rule: Displays information about the lexical rule including: the unifications associated with the lexical rule.

Word: Displays a list of senses for the word.

Sense: In versions incorporating the morphological analyzer, displays information about the sense including: a list of morphemes which were used to build the sense.

In versions without the morphological analyzer, displays information about the sense including: the unifications associated with the sense (from the lexical entry), a DAG associated with the sense, etc.

Morpheme: In versions incorporating the morphological analyzer, displays information about the morpheme including: the unifications associated with the morpheme (from the lexical entry), a DAG associated with the morpheme, etc.

In addition, information about the source file in which the definition of the object occurs is also displayed for rules, templates, and other user-defined objects.

Edit: Sends user to the editor, editing the source file in which the object associated with the icon is defined. The cursor is placed at the beginning of the definition of the object. The definition can then be edited, incrementally compiled (using the ZMACS command control-shift-C), etc. See Section B.3.1.

Trace: Occurs in the menu associated with rules only. Causes this rule to be traced (if tracing is allowed, see Section B.2.1). See Section B.3.2.

Untrace: Occurs in the menu associated with rules only. Causes this rule to be untraced. See Section B.3.2.

B.3 Other Components of the System

B.3.1 The editor interface

N.B.: The editor interface was not written solely by the author. It consists primarily of ZETALISP source code that has been modified by the author and Mabry Tyson.

The editor interface allows PATR-II grammar and lexicon development to proceed in an incremental fashion by integrating the ZMACS editor into the grammar debugging environment.

The editor is entered either by clicking right on a mouse-sensitive icon and choosing the "Edit" option, or by choosing the "Edit" command from the center menu. In the former case, the editor will be entered in a buffer with the appropriate file and the cursor will be positioned at the beginning of the definition of the object. In the latter case, the editor will be entered in the state in which it was last exited.

The editor can be put in a PATR mode, in which several commands are redefined so as to make them more useful for editing and compiling PATR-II files. This mode is entered automatically if either the mode line specifies PATR mode, or the extension on the file is ".gram", ".defs", or ".lex". The following commands are useful when in PATR mode:

control-shift-C Compiles the rule, word, template, or lexical rule at the current cursor position, updating all tables and data structures accordingly. Gives a warning if an object of the same type with the same identifier already existed (which is the normal case; the message is thus usually ignorable).

control-shift-E Same as control-shift-C.

meta-X compile region Compiles all the definitions in the region, as per control-shift-C above.

meta-X evaluate region Same as meta-X compile region.

meta-. Prompts user for an identifier for an object and finds all definitions for a loaded object with that name. Moves to a buffer with the first such definition and positions the cursor to the beginning of the definition of the object. To get to the next definition, use control-., discussed below.

control-. If several definitions of objects with the same identifier exist, control-. moves to the next definition. (A bug in ZETALISP code temporarily disables this command.)

control-meta-P Moves backward over a definitino in a file to the definition preceding it.

control-meta-N Moves forward over a definition in a file to the definition following it.

Most of the other ZMACS commands work in the normal way and can be used for editing definitions.

B.3.2 The tracing package

The tracing package allows for the user to follow the course of a parse dynamically (albeit in a limited fashion). Traces are set on grammar rules by clicking right on the rule and choosing the option "Trace". Traces are removed by the option "Untrace".

When a rule is traced, an icon of any edge built using the rule is automatically echoed in the right interaction pane. The icons are mouse-senstitive, and can therefore be clicked on to display full information in the normal way (see Section B.2.3).

The tracing of rules is mediated by the status of settings in the profile package (see Section B.2.1). Independent control of tracing passive and active edges is provided by setting profile parameters. If the "Trace building of passive edges" parameter, e.g., is set to "No", then icons are not echoed for passive edges *regardless of whether the rule involved is traced or not.* If it is set to "Yes", then icons are printed if and only if the rule is traced. Finally, if it is set to "All", then icons *are* echoed, again regardless of the tracing status of the rule.

B.4 A Session with the PATR-II Experimental System

This section documents a session with the PATR-II Experimental System in a series of snapshots of the systems performance.

The following annotations for the snapshots describe the sequence of operations leading to the screen configuration shown and discuss particular points of interest. It should be noted that the session in question was using a configuration of the system including the morphological analyzer and theorem prover, although the theorem prover connection was turned off.

1. The system has just been started and the morphological analyzer automata are being loaded. Notification of this process is being done in a temporary window.

2. The automata loading is finished.

3. The system is now in a nascent state. The user is about to click left on "Load". Note location of mouse cursor, marked with an "x". The command being picked out is highlighted with a surrounding box. Note also that documentation concerning the command is displayed in the inverse video "mouse line" at the bottom of the screen. This is in general the case when the mouse cursor is over a command in a menu or a mouse-sensitive icon.

4. Grammar loading has completed. Notifications occurred in a temporary window. The system is now ready to parse sentences.

5. The user wants to change one of the properties of the system configuration, and so is about to click on "Profile".

6. The profile men pops up, with information loaded from the user's "klaus.init file" and the user prepares to change the configuration so that active edges are not traced.

7. The change is made by clicking on the appropriate item. The user exits the profile menu and the changes are made.

8. The user is about to display the directory showing the contents of the left display windows.

9. They are all empty at this point.

10. The user is about to click left on "Tables" to show the table of the "first" relation.

11. The "first" relation is displayed.

12. Clicking middle displays the "first-inverse" relation.

13. Clicking right displays the "left-corner" relation.

14. The user clicks the "Rules" command in the upper left menu causing a list of grammar rules to be displayed in an available window.

15. Now displaying the directory indicates that the top window holds a list of grammar rules.

16. Clicking the "Templates" command displays a list of templates.

17. Clicking the "Lexical rules" command displays a list of lexical rules.

18. The directory now indicates that three windows have information in them. The order, from top to bottom, corresponds to recency, the lexical rules being the most recently exposed. The user is about to click on the third entry in the directory, thereby moving the corresponding window to the top of the stack and exposing it.

19. The grammar rules thus come into view. The user wants to display the "generic sentence" rule, so he clicks left on it.

20. The rule is displayed.

21. The user types in a sentence to have it parsed by the system.

22. The system finds a single parse for the sentence, and prints out a representation of the translation of the sentence into a logical form language.

    The user wants to see the edges coming into the final vertex in the chart, so he clicks on an icon of that vertex.

23. The vertex information is displayed in an available display window.

    The user picks an edge to display and clicks on it.

24. The edge is displayed. The user clicks on the child constituent of this edge, to display it.

25. It in turn is displayed. The user clicks on the rule that was used in forming this edge, the "generic sentence" rule.

26. The rule is displayed again. The user wants to edit this rule, so he clicks right on the icon for the rule.

27. A menu pops up allowing him to choose an operation to perform on this rule. He picks the "Edit" option.

28. The system pulls up the ZMACS editor window, loads the source file for the rule, and positions the cursor at the beginning of the rule definition, as seen in this snapshot.

29. The user adds a unification to the rule using normal ZMACS editing commands.

30. The ZMACS command control-shift-C causes the rule to be incrementally compiled into the system. A warning is given that it is replacing the old version of the rule with the new version.

    The user is about to click the mouse while the mouse cursor is over the PATR-II window, not the editor window. This will reexpose the PATR-II configuration.

31. The PATR-II window is reexposed. The user wants to see the new version of the "generic sentence" rule, so clicks on "Show".

32. From the pop up menu, he chooses "Rule".

33. The user enters the identifier for the rule he wishes to see.

34. The rule is displayed. The user wants to examine the chart (although he could as well use the chart icons appearing in the right window).

35. The chart is displayed in a display window. The user clicks on the word icon "merlyn" to display it.

36. The word has only one sense; it is chosen.

37. The information for that sense of the word is displayed. It has been derived by the morphological analyzer from two morphemes. The first one is clicked on.

38. The information for this morpheme is displayed. This information comes directly from the lexical entry in the "sept3.lex" file. The definition makes use of the template "Name". The user clicks on an icon for that template.

39. The template is displayed. The "Directory" command is about to be clicked on.

40. The directory shows the seven most recent windows of information, in order of recency. The user pulls up the chart window to the front by clicking left on it.

41. The chart window is moved to the front. Vertex 6 is about to be displayed.

42. The vertex information is displayed. The user chooses an edge.

43. The edge is displayed. The user clicks on a sense icon incorporated in the DAG associated with the edge.

44. The sense of "persuaded" used in the final parse is displayed. The user looks at the first morpheme involved in building up the sense.

45. The morpheme is displayed. The user swaps the front two windows.

46. Back at the sense window, the user clicks on the other morpheme, the "ed" ending.

47. The ending is displayed.

48. The directory shows the appropriate window contents.

49. The user is about to click on the "Rotate forward" command.

50. The directory shows that the windows have been rotated as expected. Notice that the sense window is now on top and displayed.

51. "Rotate back" works similarly.

52. The windows are moved back to their original positions.

53. The left window is refreshed (by hitting the refresh key) and a new sentence is typed in. But the user has misspelled a word. The system asks if he wants to replace the word. The user is about to click on "Yes".

54. The user enters the new word to replace the misspelling.

55. The parse continues, and a single parse is found. The user wants to reparse the sentence, so clicks on "Reparse".

56. The sentence (with misspelling corrected) is reparsed.

57. The screen is refreshed, and the user is about to clear the grammar.

58. The grammar information is cleared and a notification is put in a temporary window.

59. The user concludes the session by clicking on "Stop".

Appendix C

User Manual for the P-PATR System

This appendix was written by Susan Hirsh.

C.1 Starting Up the System

To start P-PATR, load the file LOADPATR.PL into the Prolog database[1]. This file loads the rest of the system and initializes all execution flags.

C.1.1 Loading necessary files

The P-PATR system consists of three basic modules: READPATR.PL, COMPILEPATR.PL and PATRLIBRARY.PL. Each of these modules is in turn divided further into submodules, which are loaded by their parent module. A complete list of all files that must reside in the Prolog database for compilation to proceed is given below.

READPATR.PL

This module includes all files necessary in the translation of a PATR grammar to clausal form. The files are

- READTOKENS.PL. Reads in a PATR rule and returns it as a list of tokens.
- READPATR.PL. Takes a list of tokens and translates it to clausal form.

[1] Loading a file into Prolog involves either compiling or interpreting that file. The current implementation compiles these files, but the system could easily be modified to interpret them, if desired. The difference is that it takes longer to compile than to interpret a Prolog file, but a compiled file executes much more quicklt.

COMPILEPATR.PL

This module includes all files necessary in the conversion of a clausal representation of a PATR grammar to a Prolog DCG. The files are

- COMPILEPATR.PL. Compiles a clausal form into a DCG.
- READRULES.PL. Reads in a list of PATR rules.
- PARAMETERS.PL. Records the information contained in the parameter statements.
- PATHS.PL. Compiles all information on the position and order of the features.
- EPSILONS.PL. Preprocesses all epsilon rules.
- COMPILEGRAMMAR.PL. Performs the actual compilation of the grammar entries.
- UNIFY.PL. Applies the unification equations constraining a rule.
- COMPILELEX.PL. Compiles all lexical entries.

PATRLIBRARY.PL

This module consists of one file that contains predicates common to all of the modules. The predicates included perform basic operations needed by the entire system.

C.1.2 Trace flags

In LOADPATR.PL there are four execution flags that can be toggled by the user:

- *trace_input* (default *no*). *Yes* prints out the clausal representation of each PATR rule as it is processed in the grammar input module.
- *trace_paths* (default *no*). *Yes* prints the feature information compiled during execution of the attribute position generation module.
- *trace_rules* (default *no*). *Yes* prints out each DCG rule as it is processed in the compilation module.
- *load_parser* (default *yes*). *No* suppresses the loading of the compiled DCG after compilation.

To change the values of any of the execution flags, the user must modify the values in LOADPATR.PL[2].

[2]The values can also be changed at a later time through the use of the Prolog predicates **abolish** and **assert** [3].

C.2 Compiling a PATR Grammar

Once all of the necessary files reside in the Prolog database, the system is ready to be used.

C.2.1 Grammar input

The file to be compiled must first be translated into clausal form by a call to the grammar input module. The calling sequence is

```
grammar( File ).
```

where the name of the file to be compiled can be any Prolog atom or string [3].

The grammar input module then translates the file into clausal form and puts the output into a new file whose name is that of the initial file with the new file type extension PTRP. When the input module is invoked it outputs to the screen the message

"Reading ..."

and once input is completed the execution time (in seconds) of the input module is printed.

For example, the file DEMOGRAM.PATR is translated to clausal form through the call

```
grammar( 'demogram.patr' ).
```

producing the new file DEMOGRAM.PTRP.

C.2.2 Grammar compilation

Once the PATR grammar is in clausal form, it is then compiled into a Prolog DCG by a call to the grammar compilation module. The calling sequence is

```
compilepatr( File ).
```

where the file type in the name of the file may be omitted as it is assumed to have the extension PTRP.

The grammar compilation module then compiles that file into a DCG and puts the output in a new file whose name is that of the initial file with the new file type extension DCG. When the compilation module is invoked it outputs to the screen the message:

"Compiling ..."

and once compilation is completed the execution time (in seconds) of the compilation module[3] is printed.

For example, the file DEMOGRAM.PTRP is compiled through the call

```
compilepatr( demogram ).
```

producing the new file DEMOGRAM.DCG.

C.3 Parsing a Sentence

C.3.1 Loading the DCG

Once the PATR grammar is compiled, the DCG file is loaded into the Prolog database[4]. When loaded, the DCG file itself loads a support module PATRSUPPORT.PL that contains added predicates that are needed in parsing. PATRSUPPORT.PL loads with it a submodule PP.PL that contains a feature structure pretty-printer, as well as submodule READIN.PL that contains a sentence reader. In all, the files that must reside in the Prolog database for parsing to be possible are

- File.DCG - DCG file produced by compilation module.
- PATRSUPPORT.PL - support module for the parser.
- PP.PL - feature structure pretty-printer.
- READIN.PL - sentence reader.

C.3.2 Sentence parsing

Once all necessary files are loaded, sentences can be parsed by entering the statement

```
patr.
```

The parser is now ready for input.

Sentence input

The input environment consists of an input loop for the sentences. Each sentence entered at the prompt ":" is parsed. End of input is signaled by the command "Control-d" entered at the input prompt.

[3]This is not completely accurate. The execution time of the compilation module is output to two steps. First the execution time of the compilation itself is printed, and then if the *load_parser* execution flag has been toggled on, a second execution time is printed corresponding to the loading time.

[4]This can be done by toggling an execution flag or by manually loading it into the Prolog database.

Parser output

Once a sentence is parsed, four pieces of information are returned by the parser:

- *Number of parses*

  The number of parses for the sentence is printed.

- *Execution time*

  The time (in seconds) that it took to parse the sentence is printed.

- *Parse tree*

  A parse tree is displayed for each of the parses for the sentence. The parse tree is represented as a parenthesized list.

  For example, for a sentence "Uther sleeps" the parse tree might be

  s(np(n(uther)), vp(v(sleeps)))

- *Feature structure corresponding to the sentence*

  A feature structure for each of the parses for a sentence is displayed as an attribute-value matrix.

  For example, a possible feature structure associated with the sentence "Uther sleeps" is represented as

```
[ cat: s
  head: [ form: finite
          trans: [ pred: sleep
                   arg1: uther ]
          aux: false ] ]
```

C.3.3 Sample session with the P-PATR system

The following is a transcript of a session with P-PATR using the grammar in Section C.4.

```
Quintus Prolog Release 1.6 (Sun)
Copyright (C) 1986, Quintus Computer Systems, Inc.  All rights reserved.

| ?- compile(loadpatr).
[pp.pl compiled (7.350 sec 1848 bytes)]
[readin.pl compiled (2.450 sec 964 bytes)]
[patrsupport.pl compiled (18.017 sec 5552 bytes)]
[patrlibrary.pl compiled (2.100 sec 728 bytes)]
[readtokens.pl compiled (9.634 sec 2968 bytes)]
[readpatr.pl compiled (28.716 sec 9948 bytes)]
[readrules.pl compiled (1.067 sec 432 bytes)]
[paths.pl compiled (12.717 sec 3700 bytes)]
[epsilons.pl compiled (1.317 sec 620 bytes)]
[parameters.pl compiled (1.850 sec 496 bytes)]
[compilegrammar.pl compiled (5.483 sec 1520 bytes)]
[compilelex.pl compiled (0.634 sec 244 bytes)]
[unify.pl compiled (3.950 sec 900 bytes)]
[compilepatr.pl compiled (29.833 sec 9388 bytes)]
[loadpatr.pl compiled (79.850 sec 26588 bytes)]

yes
| ?- grammar('sample.patr').

Reading ...

Runtime = 11.899994

yes
| ?- compilepatr(sample).

Compiling ...

Runtime = 5.633995

Loading ...
[sample.dcg compiled (20.633 sec 3728 bytes)]

yes
| ?- patr.
|: Uther sleeps.
```

```
Runtime = 0.066000

Analysis # 1:

Parse Tree = s(np(uther),vp(v(sleeps)))

[cat: s
 head: [form: finite
        trans: [pred: sleep
                arg1: uther]
        aux: false]]


Number of Parses = 1
|: Cornwall is stormed.

Runtime = 0.100000

Analysis # 1:

Parse Tree = s(np(cornwall),vp(vp(v(is)),vp(v(stormed))))

[cat: s
 head: [form: finite
        trans: [pred: storm
                arg2: cornwall]]]


Number of Parses = 1
|: Knights sleep.

Runtime = 0.084000

Analysis # 1:

Parse Tree = s(np(nom(knights)),vp(v(sleep)))

[cat: s
 head: [form: finite
        trans: [pred: sleep
                arg1: knights]
        aux: false]]
```

```
Number of Parses = 1
|: A knight storms Cornwall.

Runtime = 0.100000

Analysis # 1:

Parse Tree = s(np(det(a),nom(knight)),vp(vp(v(storms)),np(cornwall)))

[cat: s
 head: [form: finite
        trans: [pred: storm
                arg1: knight
                arg2: cornwall]
        aux: false]]


Number of Parses = 1
|: Uther storms Cornwall.

Runtime = 0.067000

Analysis # 1:

Parse Tree = s(np(uther),vp(vp(v(storms)),np(cornwall)))

[cat: s
 head: [form: finite
        trans: [pred: storm
                arg1: uther
                arg2: cornwall]
        aux: false]]


Number of Parses = 1
|: Uther sleep.

Runtime = 0.050000

*** Cannot parse [uther,sleep]
|: A knights storm Cornwall.

Runtime = 0.050000

*** Cannot parse [a,knights,storm,cornwall]
```

```
|: ^D
yes
| ?- halt.

[ End of Prolog execution ]
```

C.4 Sample grammar and Prolog DCG

```
;;;=====================================================
;;;                Demonstration Grammar
;;;          (adapted from Sample Grammar 4 in [14])
;;;
;;; Includes:   subject-verb agreement
;;;             complex subcategorization
;;;             logical form construction
;;;             lexical organization by templates
;;;               and lexical rules
;;;=====================================================

Parameter:  Start Symbol is S.

Parameter:  Attribute order is  cat lex sense head
                                subcat first rest
                                form agreement person
                                        number gender
                                trans pred arg1 arg2.

;;;=====================================================
;;;                    Grammar Rules
;;;=====================================================

Rule {sentence formation}

        S -> NP VP:

                <S head> = <VP head>
                <S head form> = finite
                <VP subcat first> = <NP>
                <VP subcat rest> = end.

Rule {np formation}

        NP -> Det Nom:

                <NP head> = <Det head>
                <NP head> = <Nom Head>.

Rule {plural nouns}

        Det -> :
```

```
                <Det head agreement number> = plural.

Rule {trivial verb phrase}

        VP -> V:

                <VP head> = <V head>
                <VP subcat> = <V subcat>.

Rule {complements}

        VP_1 -> VP_2 X:

                <VP_1 head> = <VP_2 head>
                <VP_2 subcat first> = <VP_1 subcat first>
                <VP_2 subcat rest first> = <X>
                <VP_2 subcat rest rest> = <VP_1 subcat rest>.

;;;=====================================================
;;;                   Definitions
;;;=====================================================

Let Verb be <cat> = v.

Let Finite be Verb
              <head form> = finite.

Let Nonfinite be Verb
                 <head form> = nonfinite.

Let ThirdPerson be <subcat first head agreement person> = third.

Let Singular be <subcat first head agreement number> = singular.

Let Plural be <subcat first head agreement number> = plural.

Let ThirdSing be Finite
                 ThirdPerson
                 Singular.

Let MainVerb be Verb
                <head aux> = false.

Let Transitive be MainVerb
```

```
                    <subcat first cat> = NP
                    <subcat rest first cat> = NP
                    <subcat rest rest> = end
                    <head trans arg1> = <subcat first head trans>
                    <head trans arg2> = <subcat rest first head trans>.

Let Intransitive be MainVerb
                      <subcat first cat> = NP
                      <subcat rest> = end
                      <head trans arg1> = <subcat first head trans>.

Let Raising be  <subcat first cat> = NP
                <subcat rest first cat> = VP
                <subcat rest first subcat rest> = end
                <subcat rest first subcat first> = <subcat first>
                <subcat rest rest> = end.

Define AgentlessPassive as <out cat> = <in cat>
                           <out subcat> = <in subcat rest>
                           <out head agreement> = <in head agreement>
                           <out head aux> = <in head aux>
                           <out head trans> = <in head trans>
                           <out head form> = passiveparticiple.

;;;=======================================================
;;;                  Lexicon
;;;=======================================================

Word uther:

           <cat> = np
           <head agreement gender> = masculine
           <head agreement person> = third
           <head agreement number> = singular
           <head trans> = uther.

Word cornwall:

           <cat> = np
           <head agreement person> = third
           <head agreement number> = singular
           <head trans> = cornwall.

Word knights:
```

```
            <cat> = nom
            <head agreement gender> = masculine
            <head agreement person> = third
            <head agreement number> = plural
            <head trans> = knights.

Word knight:

            <cat> = nom
            <head agreement gender> = masculine
            <head agreement person> = third
            <head agreement number> = singular
            <head trans> = knight.

Word a:

            <cat> = det
            <head agreement number> = singular.

Word sleeps:    Intransitive ThirdSing
                <head trans pred> = sleep.

Word sleep:     Intransitive Plural
                <head trans pred> = sleep.

Word storms:    Transitive ThirdSing
                <head trans pred> = storm.

Word stormed:   Transitive AgentlessPassive
                <head trans pred> = storm.

Word is:        Raising ThirdSing
                <subcat rest first head form> = passiveparticiple
                <head trans> = <subcat rest first head trans>.
```

The following is the DCG produced by P-PATR for the grammar presented above.

```
ensure_loaded(patrsupport).

start(s).

feature_order(main,[cat:_6688,lex:_6681,sense:_6674,head:_6660,
                    subcat:_6667],
                   [_6688,_6681,_6674,_6660,_6667]).
feature_order(head,[form:_6873,agreement:_6880,trans:_6866,aux:_6887],
                   [_6866,_6873,_6880,_6887]).
feature_order(subcat,[first:_7024,rest:_7031],
                     [_7024,_7031]).
feature_order(first,[cat:_7148,lex:_7141,sense:_7134,head:_7120,
                     subcat:_7127],
                    [_7148,_7141,_7134,_7120,_7127]).
feature_order(rest,[first:_7328,rest:_7335],
                   [_7328,_7335]).
feature_order(agreement,[person:_7431,number:_7424,gender:_7438],
                        [_7424,_7431,_7438]).
feature_order(trans,[pred:_7558,arg1:_7565,arg2:_7572],
                    [_7558,_7565,_7572]).
feature_order(arg1,[pred:_7690,arg1:_7697,arg2:_7704],
                   [_7690,_7697,_7704]).
feature_order(arg2,[pred:_7822,arg1:_7829,arg2:_7836],
                   [_7822,_7829,_7836]).

null([cat: det,
     lex: _7973,
     sense: _7978,
     head: [form: _8072,
            agreement: [person: _8117,
                        number: plural,
                        gender: _8127],
            trans: _8082,
            aux: _8087],
     subcat: _7988]).

lc([cat: np,
    lex: _8231,
    sense: _8236,
    head: _8241,
    subcat: _8246],
    _8691,_8746,_8693)-->
```

```
down([cat: vp,
      lex: _8179,
      sense: _8184,
      head: [form: finite,
             agreement: _8491,
             trans: _8496,
             aux: _8501],
     subcat: [first: [cat: np,
                      lex: _8231,
                      sense: _8236,
                      head: _8241,
                      subcat: _8246],
              rest: end]],
     _8686),
lc([cat: s,
    lex: _8283,
    sense: _8288,
    head: [form: finite,
           agreement: _8491,
           trans: _8496,
           aux: _8501],
    subcat: _8298],
    _8691,s(_8746,_8686),_8693).

lc([cat: det,
    lex: _8855,
    sense: _8860,
    head: _8813,
    subcat: _8870],
    _9156,_9211,_9158)-->
down([cat: nom,
      lex: _8803,
      sense: _8808,
      head: _8813,
      subcat: _8818],
      _9151),
lc([cat: np,
    lex: _8907,
    sense: _8912,
    head: _8813,
    subcat: _8922],
    _9156,np(_9211,_9151),_9158).

lc([cat: nom,
    lex: _8803,
```

```
    sense: _8808,
    head: [form: _9261,
           agreement: [person: _9267,
                       number: plural,
                       gender: _9269],
           trans: _9259,
           aux: _9271],
    subcat: _8818],
    _9283,_9338,_9285)-->
lc([cat: np,
    lex: _8907,
    sense: _8912,
    head: [form: _9261,
           agreement: [person: _9267,
                       number: plural,
                       gender: _9269],
           trans: _9259,
           aux: _9271],
    subcat: _8922],
    _9283,np(_9338),_9285).

lc([cat: v,
    lex: _9394,
    sense: _9399,
    head: _9404,
    subcat: _9409],
    _9687,_9742,_9689)-->
lc([cat: vp,
    lex: _9446,
    sense: _9451,
    head: _9404,
    subcat: _9409],
    _9687,vp(_9742),_9689).

lc([cat: vp,
    lex: _9798,
    sense: _9803,
    head: _9808,
    subcat: [first: _10053,
             rest: [first: _286,
                    rest: _10141]]],
    _10472,_10527,_10474)-->
down(_286,_10467),
lc([cat: vp,
    lex: _9850,
```

```
    sense: _9855,
    head: _9808,
    subcat: [first: _10053,
             rest: _10141]],
    _10472,vp(_10527,_10467),_10474).

lex(uther,[cat: np,
           lex: uther,
           sense: uther1,
           head: [form: _10838,
                  agreement: [person: third,
                              number: singular,
                              gender: masculine],
                  trans: uther,
                  aux: _10848],
           subcat: _10850]).

lex(cornwall,[cat: np,
              lex: cornwall,
              sense: cornwall1,
              head: [form: _10838,
                     agreement: [person: third,
                                 number: singular,
                                 gender: _10846],
                     trans: cornwall,
                     aux: _10848],
              subcat: _10850]).

lex(knights,[cat: nom,
             lex: knights,
             sense: knights1,
             head: [form: _10838,
                    agreement: [person: third,
                                number: plural,
                                gender: masculine],
                    trans: knights,
                    aux: _10848],
             subcat: _10850]).

lex(knight,[cat: nom,
            lex: knight,
            sense: knight1,
            head: [form: _10838,
                   agreement: [person: third,
                               number: singular,
```

```
                                gender: masculine],
                        trans: knight,
                        aux: _10848],
                subcat: _10850]).

lex(a,[cat: det,
       lex: a,
       sense: a1,
       head: [form: _10838,
              agreement: [person: _10844,
                          number: singular,
                          gender: _10846],
              trans: _10836,
              aux: _10848],
       subcat: _10850]).

lex(sleeps,[cat: v,
            lex: sleeps,
            sense: sleeps1,
            head: [form: finite,
                   agreement: _10846,
                   trans: [pred: sleep,
                           arg1: _10840,
                           arg2: _10842],
                   aux: false],
            subcat: [first: [cat: np,
                             lex: _10966,
                             sense: _10964,
                             head: [form: _10956,
                                    agreement: [person: third,
                                                number: singular,
                                                gender: _11322],
                                    trans: _10840,
                                    aux: _10960],
                             subcat: _10962],
                     rest: end]]).

lex(sleep,[cat: v,
           lex: sleep,
           sense: sleep1,
           head: [form: _10844,
                  agreement: _10846,
                  trans: [pred: sleep,
                          arg1: _10840,
                          arg2: _10842],
```

```
                aux: false],
          subcat: [first: [cat: np,
                           lex: _10966,
                           sense: _10964,
                           head: [form: _10956,
                                  agreement: [person: _11170,
                                              number: plural,
                                              gender: _11172],
                                  trans: _10840,
                                  aux: _10960],
                           subcat: _10962],
                   rest: end]]).

lex(storms,[cat: v,
            lex: storms,
            sense: storms1,
            head: [form: finite,
                   agreement: _10846,
                   trans: [pred: storm,
                           arg1: _10840,
                           arg2: _10842],
                   aux: false],
            subcat: [first: [cat: np,
                             lex: _10966,
                             sense: _10964,
                             head: [form: _10956,
                                    agreement: [person: third,
                                                number: singular,
                                                gender: _11366],
                                    trans: _10840,
                                    aux: _10960],
                             subcat: _10962],
                     rest: [first: [cat: np,
                                    lex: _10988,
                                    sense: _10986,
                                    head: [form: _10978,
                                           agreement: _10980,
                                           trans: _10842,
                                           aux: _10982],
                                    subcat: _10984],
                            rest: end]]]).

lex(stormed,[cat: v,
             lex: _10854,
             sense: _10852,
```

```
            head: [form: passiveparticiple,
                   agreement: _10846,
                   trans: [pred: storm,
                           arg1: _10840,
                           arg2: _10842],
                   aux: false],
            subcat: [first: [cat: np,
                             lex: _10988,
                             sense: _10986,
                             head: [form: _10978,
                                    agreement: _10980,
                                    trans: _10842,
                                    aux: _10982],
                             subcat: _10984],
                     rest: end]]).

lex(is,[cat: v,
        lex: is,
        sense: is1,
        head:
         [form: finite,
          agreement: _10840,
          trans: _10836,
          aux: _10842],
        subcat:
         [first:
           [cat: np,
            lex: _10984,
            sense: _10982,
            head:
             [form: _11256,
              agreement:
               [person: third,
                number: singular,
                gender: _11264],
              trans: _11254,
              aux: _11266],
            subcat: _10980],
          rest:
           [first:
             [cat: vp,
              lex: _10866,
              sense: _10864,
              head:
               [form: passiveparticiple,
```

```
     agreement: _10858,
     trans: _10836,
     aux: _10860],
   subcat:
    [first:
      [cat: np,
       lex: _10984,
       sense: _10982,
       head:
        [form: _11256,
         agreement:
          [person:third,
           number: singular,
           gender: _11264],
         trans: _11254,
         aux: _11266],
       subcat: _10980],
     rest: end]],
rest: end]]]).
```

C.5 Selected code

```
%  Module: COMPILEPATR.PL
%  Author: Susan B. Hirsh
%  Purpose: Compile a clausal form of a PATR-II grammar into a
%           DCG.

% load all supplemental modules

:- ensure_loaded( readrules ).       % read in PATR-II rules
:- ensure_loaded( parameters ).      % handle parameter statements
:- ensure_loaded( paths ).           % generate feature information
:- ensure_loaded( epsilons ).        % precompile epsilon rules
:- ensure_loaded( compilegrammar ). % compile the PATR-II grammar
:- ensure_loaded( unify ).           % unify PATR-II equations
:- ensure_loaded( compilelex ).      % precompile lexical entries



% External predicates :
%
%
% Module COMPILEGRAMMAR.PL -
%
% compile_grammar/3 -
%    compile PATR-II grammar into a DCG.
%
%
% Module COMPILELEX.PL
%
% compile_lex/1 -
%   execute each lexical entry in the database.
%
%
% Module EPSILONS.PL
%
% epsilons/2 -
%    precompile epsilon rules.
%
%
% Module PARAMETERS.PL
%
% parameter/3 -
%    process all parameter statements.
```

```
%
%
% Module PATHS.PL
%
% paths/2 -
%    generate all feature information.
%
%
% Module PATRLIBRARY.PL
%
% file_name/3 -
%    create a new file name with a new ending.
% write_clause/2 -
%    write clause to output stream in Prolog clause format.
%
%
% Module PATRSUPPORT.PL
%
% format_stats/0 -
%    output statistics on runtime.
% set_timer/0 -
%    reset runtime timer.
%
%
% Module READRULES.PL
%
% input_rules/2 -
%    Read in PATR-II rules from .PTRP file.



%-------------------------------------------------------------
%
% compilepatr( File )
%
% Input :
%   File - name of input file (must have .PTRP extension)
%
%
% Take a list of PATR-II rules produced by READPATR.PL and
%  convert them into a Definite-clause grammar (DCG).


compilepatr( File ) :-
   format( '~nCompiling ...~n', [] ),   % output current status
```

```
   input_rules( File, Rules ),          % read in grammar rules
   output_rules( File, Rules ).         % convert rules to DCG


%-----------------------------------------------------------
%
% output_rules( File, Rules )
%
% Input :
%   File - name of input file
%   Rules - list of PATR-II rules
%
%
% Convert PATR-II rules into a DCG and output the DCG.


output_rules( File, Rules ) :-
   file_name( File, ".dcg" , Output ),   % output file is File.dcg
   open( Output, write, OutStream ),     % open output file
   % insert line into DCG to include runtime support
   write_clause( ( :- ensure_loaded( patrsupport ) ),
                 OutStream ),
   compile_rules( Rules, OutStream ),    % compile PATR-II rules
   close( OutStream ),                   % close output file
   ( load_parser( yes ) ->               % is DCG to be loaded
   load_dcg( Output )                    % load the DCG
   | true ).                             % do nothing


%-----------------------------------------------------------
%
% compile_rules( Rules, OutStream )
%
% Input :
%   Rules - list of PATR-II rules
%   OutStream - current output stream
%
%
% Compile PATR-II rules into a DCG.


compile_rules( Rules, OutStream ) :-
   set_timer,                                % set runtime timer
```

```
    parameter( Rules, OnlyRules, OutStream ), % handle parameters
    paths( OnlyRules, OutStream   ),          % get feature information
    % precompile epsilon rules
    epsilons( OnlyRules, OutStream ),
    compile_grammar( OnlyRules, Rules, OutStream ),% make DCG
    % execute lexical entries
    compile_lex( OutStream ),
    format_stats.                             % output compile statistics


%------------------------------------------------------------
%
% load_dcg( Output )
%
% Input :
%   Output - name of output file
%
%
% Load DCG into Prolog database.


load_dcg( Output ) :-
    format( '~nLoading ...~n',[] ),% output current status
    ensure_loaded( Output ).
```

```
%  Module: COMPILEPATR.PL
%  Submodule: READRULES.PL
%  Author: Susan B. Hirsh
%  Purpose: Read in a list of PATR rules.


% External predicates :
%
%
% Module PATRLIBRARY.PL
%
% file_name/3 -
%    create a new file name with a new ending.



%-----------------------------------------------------------
%
% input_rules( File, Rules )
%
% Input :
%   File - input file name
%
% Output :
%   Rules - list of all PATR-II rules from input file
%
%
% Read in PATR-II rules from input file and put into a list.


input_rules( File, Rules ) :-
   seeing( Infile ),                    % save current input file
   file_name( File, ".ptrp", Input ),   % input file is File.ptrp
   see( Input ),                        % open input file
   read_rules( Rules ),                 % read in the rules
   seen,                                % close input file
   see( Infile ).                       % restore input file



%-----------------------------------------------------------
%
% read_rules( Rules )
%
```

```
% Output :
%   Rules - list of PATR-II rules
%
% Read in a list of PATR-II rules.


read_rules( Rules ) :-
   read( Rule ),                      % read in the first rule
   read_more_rules( Rule, Rules ).    % read in the rest



%-------------------------------------------------------------
%
% read_more_rules( PreviousRules, NewRules )
%
% Input :
%   PreviousRules - list of PATR-II rules as it is being
%                   built up
%
% Output :
%   NewRules - list of PATR-II rules
%
% Read in a list of PATR-II rules.


%  stop at the end of the file
read_more_rules( end_of_file, [] ) :- !.

%  keep reading until the end of the file
read_more_rules( Rule,[ Rule | Rules ] ) :-
   read( NewRule ),                    % read in a PATR-II rule
   read_more_rules( NewRule, Rules ). % read in the rest
```

```
%  Module: COMPILEPATR.PL
%  Submodule: PARAMETERS.PL
%  Author: Susan B. Hirsh
%  Purpose: Record the information from the parameter statements.


% External predicates :
%
%
% Module PATRLIBRARY.PL
%
% write_clause/2 -
%    write clause to output stream in Prolog clause format.



%-------------------------------------------------------------
%
% parameter( Rules, NewRules )
%
% Input :
%   Rules - list of PATR-II rules
%
% Output:
%   NewRules - list of PATR-II rules minus parameter statements
%
%
% Handle parameter statements first as they must only appear at
%  the top of the file.


% handle start symbol
parameter( [ parameter( start( Symbol ) ) | Rules ], NewRules,
           OutStream ):-
   assert( start(Symbol) ),          % assert start symbol
   write_clause( ( start(Symbol) ), OutStream ),   % write to output
   parameter( Rules, NewRules, OutStream ).        % handle others

% keep track of attribute order
parameter( [ parameter( attributes( List ) ) | Rules ], NewRules,
           OutStream ):-
   % record the correct order
   record_order( List, 1 ),
   % handle other parameter stmnts
```

```
   parameter( Rules, NewRules, OutStream).

% ignore restrictor
parameter( [ parameter( restrictor( _List ) ) | Rules ], NewRules,
           OutStream ) :-
   parameter( Rules, NewRules, OutStream ).

% ignore translation
parameter( [ parameter( translation( _List ) ) | Rules ], NewRules,
           OutStream ) :-
   parameter( Rules, NewRules, OutStream ).

% no more parameter statements - return list minus parameters
parameter( Rules, Rules, _OutStream ).



%-------------------------------------------------------------
% record_order( Attributes, Place )
%
% Input :
%   Attributes - list of attributes in the order they are to appear
%   Place - position in the list of the current attribute
%
%
% Record the print order of each attribute.


% record the position of each attribute
record_order( [ Attribute | Attributes ], Place ) :-
   % assert for use in printing
   assert( print_order(Attribute,Place) ),
   % increment position
   NewPlace is Place + 1,
   % go on to the next attribute
   record_order( Attributes, NewPlace ).

% no more attributes
record_order( [], _Place ).
```

```
%  Module: COMPILEPATR.PL
%  Submodule: PATHS.PL
%  Author: Susan B. Hirsh
%  Purpose: Compile all information on the position and order of
%           the features.


% External predicates :
%
%
% Module PATRLIBRARY.PL
%
% write_clause/2 -
%    write clause to output stream in Prolog clause format.
%
%
% Module PATRSUPPORT.PL
%
% print_order/2 -
%    the printing order of this feature in the feature structure.



%-----------------------------------------------------------
%
% paths( Rules, OutStream )
%
% Input :
%   Rules - list of PATR-II rules
%   OutStream - current output stream
%
%
% Generate for each attribute a list of which features can
%  follow it and assert this information into the database
%  and output into output file.
%
% For example :
%
%  The rule
%   rule(NP,[N],[[NP,cat]=np,[N,cat]=n,[NP,body]=[N,body]])
%
%  would produce the list :
%     feature_order(main,[cat:X,body:Y],[X,Y])
%
```

```
%  where the attribute 'main' is a dummy attribute used to designate
%  that a feature that follows it was the first feature in a path
%  specification.

paths( Rules, OutStream ) :-
   % create lists of Vars, Bindings, and Pairs
   type_info( Rules, [ Main ], Types,[ main=Main ], Bindings, [],
              Pairs ),
   calc_types( Pairs ),      % make pairs into paths
   tails( Types ),           % get rid of tail variables
   % assert paths into database and write into output file
   output_paths( Bindings, OutStream ).



%-------------------------------------------------------------
%
% type_info( Rules, OldTypes, Types, OldBindings, Bindings,
%            OldPaths, Paths )
%
% Input :
%   Rules - list of rules in PATR-II format
%   OldTypes - types found so far
%   OldBindings - bindings found so far
%   OldPaths - paths found so far
%
% Output :
%   Types - list of types
%   Bindings - list of bindings
%   Paths - list of paths
%
%
% Extract from each rule the features used in that rule.  From
%  this feature information compile three different lists :
%
%  Types : a list of variables associated with the features
%  Bindings : a list containing information on which attributes
%   are bound to which variables.
%  Pairs : a list of which features can follow which others


% no more rules
type_info( [], Types, Types, Bindings, Bindings, Pairs, Pairs ).

% extract info from each rule
```

```
type_info( [ Rule | Rules ], Types, Rtypes, Bindings, Rbindings,
           Pairs, Rpairs):-
   % features are contained in the unification equations of a rule
   unifs( Rule, Unifs, Type ),   % get feature information
   % process the feature information
   info( Type, Unifs, Types, Ntypes, Bindings, Nbindings, Pairs,
         Npairs ),
   % do the rest of the rules
   type_info( Rules, Ntypes, Rtypes, Nbindings, Rbindings, Npairs,
              Rpairs).



%----------------------------------------
%
% unifs( Rule, Unifs, Type )
%
% Input :
%   Rule - current PATR-II rule
%   Type  - what kind of rule is this
%
% Output :
%   Unifs - list of unifications for that rule
%
%
% Extract the unification equations from the rule.


% grammar rule
unifs( rule(_Lhs,_Rhs,Unifs), Unifs, rule ).

% lexical entry
unifs( lex(_Word,Unifs), Unifs, lex ).

% lexical template
unifs( template(_Name,Unifs), Unifs, lex ).

% lexical rule
unifs( lex_rule(_Name,_InFS,_OutFS,Unifs), Unifs, rule ).



%-----------------------------------------------------------
% info( Type, Unifs, OldTypes, Types, OldBindings, Bindings,
%       OldPaths, Paths )
```

```
%
% Input :
%   Type - the type of rule it is
%   Unifs - list of unifications for that rule
%   OldTypes - list of types so far
%   OldBindings - list of bindings so far
%   OldPaths - list of paths so far
%
% Output :
%   Types - list of types
%   Bindings - list of bindings
%   Paths - list of paths
%
%
% Extract feature information from the unification equations.


% no more unifications in this rule
info( _All, [], Types, Types, Bindings, Bindings, Pairs,
      Pairs ).

% ignore template and lexical rule names, as these features are
%  handled in template or rule definitions
info( Kind, [ Template| T ], Types, Rtypes, Bindings, Rbindings,
      Pairs, Rpairs) :-
   atomic( Template ), !,   % this is a template or lexical rule
   % go on to the next unification equation
   info( Kind, T, Types, Rtypes, Bindings, Rbindings, Pairs,
         Rpairs ).

% for rules :

%   handle unifications of the form : Path1 = Path2
%    E.G.,
%      <S head> = <VP head>
info( rule, [ [ _Var1 | Features1 ] =
              [ _Var2 | Features2 ] | T ],
   Types, Rtypes, Bindings, Rbindings, Pairs, Rpairs ) :-
   % unify the final feature values so that paths can unify
   add_paths( Features1, Types, Ntypes, Bindings, Nbindings, Pairs,
              Npairs, main, Last),
   add_paths( Features2, Ntypes, Mtypes, Nbindings, Mbindings,
              Npairs, Mpairs, main, Last ),
   % go on to the next unification equation
   info( rule, T, Mtypes, Rtypes, Mbindings, Rbindings, Mpairs,
```

```
        Rpairs ).

%   handle unifications of the form : Path = val
%    E.G.,
%      <X cat> = np
info( rule, [ [ _Var | Features ]=Atom | T ], Types, Rtypes,
      Bindings, Rbindings, Pairs, Rpairs ) :-
   atomic( Atom ),
   % add feature information
   add_paths( Features, Types, Ntypes, Bindings, Nbindings, Pairs,
              Npairs, main, _Last),
   % go on to next unification
   info( rule, T, Ntypes, Rtypes, Nbindings, Rbindings, Npairs,
         Rpairs ).

% for lexical entries or templates :

%   handle unifications of the form : Path = val
%    E.G.,
%      <cat> = np

info( lex, [ Features=Atom |T ], Types, Rtypes, Bindings,
      Rbindings, Pairs, Rpairs) :-
   atomic( Atom ),!,
   % add feature information
   add_paths( Features, Types, Ntypes, Bindings, Nbindings, Pairs,
              Npairs, main, _Last ),
   % go on to next unification
   info( lex, T, Ntypes, Rtypes, Nbindings, Rbindings, Npairs,
         Rpairs ).

%   handle unifications of the form : Path1 = Path2
%    E.G.,
%      <head> = <head>
info( lex, [ Features1=Features2 | T ], Types, Rtypes, Bindings,
      Rbindings, Pairs, Rpairs ) :-
   % unify the final feature values so that paths can unify
   add_paths( Features1, Types, Ntypes, Bindings, Nbindings, Pairs,
              Npairs, main, Last ),
   add_paths( Features2, Ntypes, Mtypes, Nbindings, Mbindings,
              Npairs, Mpairs, main, Last ),
   info( lex, T, Mtypes, Rtypes, Mbindings, Rbindings, Mpairs,
         Rpairs).
```

```
%-----------------------------------------------------------
%
% add_paths( Features, OldTpes, Types, OldBindings, Bindings,
%            Oldpairs, Pairs, Place, Last )
%
% Input :
%   Features - list of features in one equation of unification
%   OldTypes - list of types so far
%   OldBindings - list of bindings so far
%   OldPairs - list of pairs so far
%   Place - previous feature
%   Last - Var value of last feature on the list
%
% Output :
%   Types - list of types
%   Bindings - list of bindings
%   Pairs - list of pairs
%
% Create the three list of Types, Bindings and Pairs as described.


% last feature, just return variable for later unifications
add_paths( [], Types, Types, Bindings, Bindings, Pairs, Pairs,
           Place, Last ):-
   % get variable equivalence of this attribute
   search( Place, Bindings, Last ).

% add on the Types, Bindings and Pairs
add_paths( [ Feature | Features ], Types, Rtypes, Bindings,
           Rbindings, Pairs, Rpairs, Place, Last ) :-
   % get variable value
   search( Place, Bindings, Var ),
   % get Type and Binding information
   checkpaths( Feature, Types, Ntypes, Bindings, Nbindings ),
   % get pair information
   add_pairs( Var, Feature, Pairs, Npairs ),
   % handle next attribute
   add_paths( Features, Ntypes, Rtypes, Nbindings, Rbindings, Npairs,
              Rpairs, Feature, Last ).



%-----------------------------------------------------------
%
```

```
% search( Place, Bindings, Var )
%
% Input :
%   Place - current attribute to look up
%   Bindings - list of bindings
%
% Output :
%   Var - Prolog Var value of the attribute
%
%
% Look up the Var value of the current attribute on the Bindings
%  list.


%  stop when you find the attribute
search( Place, [ Place=Var | _Bindings ], Var ) :- !.

%  keep searching until you find it
search( Place, [ _Binding | Bindings ], Var ) :-
   search( Place, Bindings, Var ).



%-------------------------------------------------------------
%
% checkpaths( Feature, Oldtypes, Types, Oldbindings, Bindings )
%
% Input :
%   Feature - current attribute
%   OldTypes - list of types
%   OldBindings - list of bindings
%
% Output :
%   Types - new list of types if attribute was added
%   Bindings - new list of bindings if attribute was added
%
%
% Check if a attribute is bound in the Bindings list and add it
%  if it isn't already there.


%  add attribute if it is not there
checkpaths( Feature, Types, [ Var | Types ], [],
            [ Feature=Var ] ).
```

```
%  if it is there do nothing
checkpaths( Feature, Types, Types, [ Feature=Var | Bindings ],
            [ Feature=Var | Bindings ] ) :- !.

%  if it is not there, keep trying the rest of the list
checkpaths( Feature, Types, Rtypes, [ Binding | Bindings ],
            [ Binding | Rbindings ] ) :-
   checkpaths( Feature, Types, Rtypes, Bindings, Rbindings ).



%-------------------------------------------------------------
% add_pairs( Var, Feature, OldPairs, Pairs )
%
% Input :
%   Var - var to add
%   Feature - attribute to add
%   OldPairs - previous list of pairs
%
% Output :
%   Pairs - new list of pairs
%
% Add a Var and a Feature to Pairs list.


add_pairs( Var, Feature, Pairs, [ Var : Feature | Pairs ] ).



%-------------------------------------------------------------
%
% calc_types( Pairs )
%
%  Input :
%   Pairs - list of pairs
%
%
% Once all of the Pairs have been done, go throught the Pairs list
%  and add all pairs to the one preceding them.
%
% For  example :
%  Pairs will look like :  [[A:head],[A,cat]]
%   and now A will look like [head,cat]
```

```
% add pair to list
calc_types( [ Type : Label | Pairs ] ):-
   insert( Label, Type ),   % unify it into the Prolog variable
   calc_types( Pairs ).     % go to next pair

% no more pairs
calc_types( [] ).



%-------------------------------------------------------------
% insert( Feature, Variable )
%
% Input :
%   Feature - current attribute
%   Variable - variable to insert value into
%
%
% Unify a feature into the Prolog variable if it is not already
%  there.


% it is already there, do nothing
insert( Label, [ Label | _ ] ) :- !.

% unify feature into variable
insert( Label, [ _ | Labels ] ) :-
   insert( Label, Labels ).



%-------------------------------------------------------------
%
% tails( Types )
%
% Input :
%   Types - list of types
%
% Change all tail variables, that are side-effect of Insert,
%  to [].


%  change tail variable to []
tails( Var ) :-
   var( Var ), !,      % this is a tail variable
```

```
   Var = [].

%  not a list, do nothing
tails( Atom ) :-
   atomic( Atom ), !.

%  check all internal lists
tails( [ Head | Tail ] ) :-
   tails( Head ),      % process first list
   tails( Tail ).      % process the rest of the list

tails( [] ).



%-------------------------------------------------------------
%
% output_paths( Bindings )
%
% Input :
%   Bindings - list that now has feature and all that can
%              follow it
%
%
% Go through the list of bindings that now have all features in
%  the variable and create paths.
% For example :
%  Bindings will be main=[head,cat]
%  Path is [main,[head:A,cat:B],[A,B]]


% no more in bindings list
output_paths( [], _OutStream ).

% make paths for all bindings
output_paths( [ Binding | Bindings ], OutStream ) :-
   make_path( Binding, OutStream),        % make the path
   output_paths( Bindings, OutStream ).   % go to next binding



%-------------------------------------------------------------
%
% make_path( Feature, OutStream )
%
```

```
% Input :
%   Feature - a feature and list that can follow
%   OutStream - current output stream
%
%
% Change path into Prolog variables and then assert and output.


% feature cannot be followed
make_path( _Head=[], _OutStream ).

% process this path
make_path( Head=Features, OutStream ) :-
   % change path into variables
   change( Head, Features, LabelList, VarList ),
   % assert and output
   write_path( Head, LabelList, VarList, OutStream ).



%-------------------------------------------------------------
%
% change( Head, Features, LabelList, VarList )
%
% Input :
%   Head - starting attribute
%   Features - list of features that can follow it
%
% Output :
%   LabelList - list of Prolog variables and features for the path
%   VarList - same as LabelList with no attributes
%
%
% Put path into Prolog variables
% Take binding
%    main = [cat,head]
%  and make the lists :
%    LabelList - [cat:Cat,head:Head]
%    VarList - [Cat,Head]


% no more paths
change( _Head, [], [], [] ).

% change each path
```

```
change( Head, [ Feature | Features ],[ Feature : Var | Labels ],
        [ Var | Vars ] ) :-
   change( Head, Features, Labels, Vars ).



%-------------------------------------------------------
%
% write_path( MainFeature, LabelList, VarList, OutStream )
%
%  Input :
%   MainFeature - feature that others follow
%   LabelList - list of attributes and variables
%   VarList - same as LabelList with no attributes
%   OutStream - current output stream
%
%
% Output path as:
%
%   feature_order(MainFeature, LabelList, VarList)


write_path( Main, LabelList, VarList,OutStream) :-
   % reorder the list into printing order
   reorder( LabelList, OrderLabelList ),
   % output to screen if trace flag is on
   (trace_paths( yes ) ->
   format( 'Path is ~w~n',
           [ feature_order(Main,OrderLabelList,VarList) ] )
   | true ),
   % assert into database for use during compilation
   assert( feature_order(Main,OrderLabelList,VarList) ),!,
   % write into output file for use during parse
   write_clause((feature_order(Main,OrderLabelList,VarList)),
                 OutStream ).



%-------------------------------------------------------
%
% reorder( FS, NewFS )
%
%  Input :
%   FS - feature structure to be reordered
%
```

```
%  Output :
%   NewFS - feature structure with features as specified
%
%
% Reorder the features in the FS according to that given in the
%  parameter statement.


reorder( Pairs, NewPairs ) :-
   % attach the order in which they should appear
   number( Pairs, NumberedPairs ),
   keysort( NumberedPairs, SortedPairs ),   % sort by position
   % get rid of position numbers
   clean( SortedPairs, NewPairs ).



%-------------------------------------------------------
%
% number( FS, NewFS )
%
%  Input :
%   FS - feature structure to be reordered
%
%  Output :
%   NewFS - feature structure with features labeled with
%           their position number
%
%
% Attach onto each feature in the FS the position number specified.


% number each feature
number( [ Label : Value | Rest ],
        [ Position-(Label:Value) | NRest ] ):-
   find_order( Label, Position ),   % get position number
   number( Rest, NRest ).           % do the rest

% no more features
number( [], [] ).



%-------------------------------------------------------
%
```

```
% find_order( Feature, Position )
%
%  Input :
%   Feature - feature to get position of
%
%  Output :
%   Position - position number of the feature
%
%
% Get the position number of the feature


% order was specified
find_order( Label, Position ) :-
   print_order( Label, Position ), !.   % specified order

% no order given, tack onto the end
find_order( _Label, 9999 ).



%-------------------------------------------------------------
%
% clean( FS, NewFS )
%
%  Input :
%   FS - feature structure with feature positions attached
%
%  Output :
%   NewFS - feature structure without feature positions
%
%
% Remove position numbers attached to the features.


% get rid of position number
clean( [ _N-(Label:Value) | Nrest ],
       [ Label : Value | Rest ] ) :-
   clean( Nrest, Rest ).

% no more features
clean( [], [] ).
```

```
%  Module: COMPILEPATR.PL
%  Submodule: EPSILONS.PL
%  Author: Susan B. Hirsh
%  Purpose: Preprocess all epsilon rules.


% External predicates :
%
%
% Module UNIFY.PL
%
% apply_rule_unifs/1 -
%    apply the unification equations for a grammar or lexical rule.
%
%
% Module PATRLIBRARY.PL
%
% write_clause/2 -
%    write clause to output stream in Prolog clause format.



%-------------------------------------------------------------
%
% epsilons( Rules, NewRules, OutStream )
%
% Input :
%   Rules - List of PATR-II rules
%   OutStream - current output stream
%
% Output :
%   NewRules - List of PATR-II rules minus epsilon rules
%
%
% Precompile epsilon rules for use in compilation.


% no more rules
epsilons( [], _OutStream ).

% go through the rules
epsilons( [ Rule | Rules ] , OutStream ) :-
   e_rule( Rule, OutStream ),      % is it an epsilon rule?
   epsilons( Rules, OutStream ).   % go to next rule
```

```
%-----------------------------------------------------------
%
% e_rule( Rule, OutStream )
%
% Input :
%   Rule -  PATR-II rule
%   OutStream - current output stream
%
%
% Compile the epsilon rule into a Prolog clause.
%  The clause is of the form :
%   null( FS )
%  where FS is the feature structure associated with the rule.


%  this is an epsilon rule
e_rule( rule(Lhs,[],Unifs), OutStream ) :-
   apply_rule_unifs( Unifs ), !,   % unify equations
   % output to screen if trace flag is on
   ( trace_rules( yes ) ->
   format( 'EPSILON Rule is ~w~n', [ null(Lhs) ] )
   | true ),
   % assert into the database for use during compilation
   assert( null(Lhs) ), !,
   % output to file.dcg
   write_clause( ( null(Lhs) ), OutStream ).

% error - cannot compile this rule
e_rule( rule(Lhs,[],Unifs), _OutStream ) :-
   format('~n*** Cannot compile rule: ~w~n',[rule(Lhs,[],Unifs)]).

%  not an epsilon rule
e_rule( _Rule, _OutStream ).
```

```
%  Module: COMPILEPATR.PL
%  Submodule: COMPILEGRAMMAR.PL
%  Author: Susan B. Hirsh
%  Purpose: Perform the actual compilation of the grammar entries.


% External predicates :
%
%
% Module UNIFY.PL -
%
% apply_lex_unifs/5 -
%    apply the unification equations for a template or lexical
%    entry.
% apply_rule_unifs/1 -
%    apply the unification equations for a grammar or lexical rule.
%
%
% Module PATRLIBRARY.PL
%
% clausify/3 -
%    create Prolog clause from a head and a list of clauses.
% reverse/3 -
%    reverse a list.
% write_clause/2 -
%    write clause to output stream in Prolog clause format.
%
%
% Module PATRSUPPORT.PL
%
% find_category/2 -
%    find the value of the category attribute in a feature structure.
% null/1 -
%    precompiled epsilon rule.



%-------------------------------------------------------------
%
% compile_grammar( Rules, RuleList, OutStream )
%
% Input :
%   Rules - list of rules to be made into DCG
%   RuleList - list of rules in current PATR-II grammar
```

```
%   OutStream - current output stream
%
%
% Take each PATR-II rule and convert it into a DCG rule.


% no more rules to compile
compile_grammar( [], _RuleList, _OutStream ).

% compile each rule
compile_grammar( [ Rule | Rules ], RuleList, OutStream ) :-
   % compile the rule
   compile_rule( Rule, RuleList, OutStream ),
   % do next rule
   compile_grammar( Rules, RuleList, OutStream ).



%-----------------------------------------------------------
%
% compile_rule( Rule, RuleList, OutStream )
%
% Input :
%   Rule - current PATR-II rule
%   RuleList - list of all rules in current PATR-II grammar
%   OutStream - current output stream
%
%
% Convert each PATR-II rule into a DCG rule or a Prolog clause.
%  Grammar rules become DCG rules and lexical items become
%  directly executable Prolog clauses that are executed once
%  compilation is completed, giving full lexical entries.


% ignore epsilon rules as they were precompiled
compile_rule( rule( _Lhs, [], _Unifs ), _Rules, _OutStream ).

% error - parameter statements must be at start of the file
compile_rule( parameter(_Statement), _Rules, _OutStream ) :-
   format('~n*** Parameter statements must occur at start of grammar file!~n',[]).

% handle grammar rules
% grammar rules are compiled into DCG rules of the form :
%
%   lc( Rhs1, Parent, OldBranch, NewBranch ) -->
```

```
%       down( Rhs2, Branch2 ),...down( RhsN, BranchN ),
%       lc( Lhs, Parent, Tree, NewBranch ).
%
% where:
%   Rhs1..RhsN - element of the right-hand side of the rule
%   Parent - variable associated with the parent of the rule
%   OldBranch - parse tree so far
%   NewBranch - parse tree after application of this rule
%   Branch2..BranchN - parse trees for each node
%   Tree - parse tree for that rule
%
compile_rule( rule( Lhs, Rhs, Unifs ), _Rules, OutStream ):-
   apply_rule_unifs( Unifs ),       % unify equations
   % create the DCG rule for the initial rule
   grammar_rule( Lhs, [ Rhs ], OutStream ),
   % return new list of rules with epsilon expansions
   epsilon( Rhs, AllRhs ),
   % create the DCG rule for the grammar rules
   grammar_rule( Lhs, AllRhs, OutStream ).

% handle lexical rules
% lexical rules are compiled into Prolog clauses of the form :
%
%   lex_rule( Name, InFS, OutFS ).
%
% where:
%   Name - name of this lexical rule
%   InFS - input feature structure to this rule application
%   OutFS - output feature structure after rule application
%
compile_rule( lex_rule( Name, InFS, OutFS, Unifs ), _Rules,
              _OutStream ):-
   apply_rule_unifs( Unifs ),              % unify equations
   assert( lex_rule(Name,InFS,OutFS) ).    % assert into database

% handle lexical entries
% lexical entries are compiled into clauses of the form:
%
%   word( Name, FS ) -->
%     applications of lexical rules and templates into
%     FS1..FSN, where last application puts result
%     into FS.
%
compile_rule( lex( Word, Unifs ), Rules, _OutStream ) :-
   % unify equations
```

```
   apply_lex_unifs( Unifs, Rules, List, HeadFS, _FS ),
   % put into clause form
   clausify( word(Word,HeadFS), List, Clause ),
   assert( Clause ).

% handle lexical templates
% lexical templates are compiled into clauses of the form :
%
%   template( Name, InFS, OutFS ) -->
%     applications of lexical rules and templates into
%     FS1..FSN, where last application puts result
%     into OutFS.
%
compile_rule( template( Name, Unifs ), Rules, _OutStream ) :-
   % unify equations
   apply_lex_unifs( Unifs, Rules, List, OutFS, InFS ),
   % put rule into clause form
   clausify( template(Name,InFS,OutFS), List, Clause ),
   assert( Clause ).

% rule could not be compiled - error
compile_rule( Rule, _Rules, _OutStream ) :-
   format('~n*** Cannot compile rule: ~w~n',[Rule]).



%-------------------------------------------------------------
%
% grammar_rule( Lhs, Rhs, OutStream )
%
%  Input :
%   Lhs - left-hand side of the rule
%   Rhs - all possible Rhs for this rule
%   OutStream - current output stream
%
%
% Take each possible Rhs for the rule and create a DCG rule for
%  it.


% make a DCG from each Rhs
grammar_rule(Lhs,[ [ Rhs1 | RhsN ] | MoreRhs ],OutStream) :-
   % create right-hand side
   rhs( Lhs, RhsN, Parent, [], Clauses, Branch, NewBranch),
   start_symbol( Lhs, OutStream ),      % find grammar start symbol
```

```
    % output new DCG rules to the screen if trace flag is set
    ( trace_rules( yes ) ->
    format( 'GRAMMAR RULE is ~w~n',
          [(lc(Rhs1,Parent,Branch,NewBranch) --> Clauses)])
    | true ),
    % output new DCG rule to file.dcg
    write_clause( (lc(Rhs1,Parent,Branch,NewBranch)-->Clauses),
                  OutStream ),
    % go to next right-hand side
    grammar_rule( Lhs, MoreRhs, OutStream ).

% no more right-hand sides to make rules from
grammar_rule( _Lhs, [], _OutStream ).



%-------------------------------------------------------------
%
% rhs( Lhs, Rhs, Parent, Branch, List, OldBranch, NewBranch )
%
% Input :
%   Lhs - left hand side of the rule
%   Rhs - All but first of right hand side of the rule
%   Parent - parent of this rule
%   OldBranch - branch variable for the left-hand side of the rule
%   NewBranch - branch variable for the left-hand side of the rule
%
% Output :
%   List - list of Rhs elements in the form for an LC rule
%   Branches - list of branches as variables in the parse tree
%
%
% Create the clauses for the right-hand side of the DCG rule


% keep track of branches and build up right-hand side
rhs( Lhs, [ Rhs1 | Rhs ], Parent, Branches, (down(Rhs1,Branch),NewRhs),
     OldBranch, NewBranch ) :-
   rhs( Lhs, Rhs, Parent, [ Branch | Branches ], NewRhs, OldBranch,
        NewBranch ).

% no more branches, create left-hand side
rhs( Lhs, [], Parent, Branches, lc(Lhs, Parent, Tree, NewBranch),
     OldBranch, NewBranch ) :-
   % get category for parse tree
```

```
   find_category( Lhs, Cat ),
   % put constituents in proper order for tree
   reverse( Branches, [], Constituents ),
   % put into form Cat(OldBranch,Constituents)
   Tree =..[Cat|[OldBranch|Constituents]].



%-------------------------------------------------------------
%
% epsilon( Rule, Newrules )
%
%  Input :
%   Rule - current rule
%
% Output :
%   Newrule - a list of all possible right-hand sides for this
%             rule
%
%
% As long as the first element of the right-hand side of the rule
%  can be expanded by an epsilon rule, return the rule minus that
%  element.
%
% For example :
%   The rules   S -> NP VP
%               NP -> e
%
% Will produce the list [ VP ], since the NP can be expanded by
%  the epsilon rule.  When returned, there are understood
%  to be two rules now instead of the one.  The rules are:
%
%      S -> NP VP
%      S -> VP


% check the first element of the right-hand side
epsilon( [ Rhs1 | Rhs ], [ Rhs | NewRhs ]) :-
   null( Rhs1 ), !,        % can be expanded by an epsilon rule
   epsilon( Rhs, NewRhs ). % check if next can be

% no more non-terminals can be expanded by epsilon rule
epsilon( _Rhs, [] ).
```

```
%-------------------------------------------------------------
%
% start_symbol( Lhs, OutStream )
%
% Input :
%   Lhs - left-hand side of the rule
%   OutStream - current output stream
%
%
% If no start symbol for the grammar has been specified, it is
%  the non-terminal on the left-hand side of the first rule.


% start symbol is already specified
start_symbol( _Lhs, _OutStream ) :-
   start( _Cat ), !.      % start symbol is in database

% no start symbol, need to add one
start_symbol( Lhs, OutStream ) :-
   find_category( Lhs, Cat ),   % get Cat of start symbol
   assert( start(Cat) ),        % assert as start symbol
   % start symbol is needed in parsing, so output to parser file
   write_clause( ( start(Cat) ) , OutStream ).
```

```
%  Module: COMPILEPATR.PL
%  Submodule: UNIFY.PL
%  Author: Susan B. Hirsh
%  Purpose: Apply the unification equations constraining a rule.


% External predicates :
%
%
% Module PATRSUPPORT.PL
%
% thepath/4 -
%   extract the value for a particular feature from a feature structure.



%-----------------------------------------------------------
%
% apply_rule_unifs( Unifs )
%
% Input :
%   Unifs - list of unifications for the rule
%
%
%  Unify the values in each equation in a grammar or lexical rule.


%  handle unifications of the form : Path1 = Path2
%   E.G.,
%    <S head> = <VP head>
apply_rule_unifs( [ [ Var1 | Features1 ]=
                    [ Var2 | Features2 ] | T ] ) :-
   % find paths from these features and unify values
   find_path( main, Features1, Val, Var1 ),
   find_path( main, Features2, Val, Var2),
   apply_rule_unifs( T ).

%  handle unifications of the form : Path = val
%   E.G.,
%    <X cat> = np
apply_rule_unifs( [ [ Var | Features ]=Atom | T ] ):-
   atomic( Atom ),
   % find location of this feature
   find_path( main, Features, Atom, Var ),
```

```
   apply_rule_unifs( T ).

% no more unification equations
apply_rule_unifs( [] ).



%-------------------------------------------------------------
%
% apply_lex_unifs( Unifs, Rules, List, HeadFS, FS )
%
% Input :
%   Unifs - list of unifications for the rule
%   Rules - list of rule in the grammar
%
%  Output :
%   List - list of rules to assert
%   HeadFS - feature structure for head of clause
%   FS - initial input feature structure
%
%
%  Unify the values in each equation in a lexical entry or template.


% no more unifications to do
apply_lex_unifs( [], _Rules, [], FS, FS ).

%  handle unifications of the form : Path = val
%   E.G.,
%    <X cat> = np
apply_lex_unifs( [ Features=Atom | T ], Rules, List, HeadFS,
                 FS ):-
   atomic( Atom ),!,
   % get position of this feature
   find_path( main, Features, Atom, FS ),
   apply_lex_unifs( T, Rules, List, HeadFS, FS ).

% add application of a lexical template to the new DCG rule
apply_lex_unifs( [ Atom | T ], Rules,
                 [ template(Atom,FS,TempFS) | List ],
                 HeadFS, FS ) :-
   atomic( Atom ),
   % make sure this is a template
   find_type( Atom, Rules, template ),!,
   apply_lex_unifs( T, Rules, List, HeadFS, TempFS ).
```

```
% add application of a lexical rule to the new DCG rule
apply_lex_unifs( [ Atom | T ] , Rules,
                 [ lex_rule(Atom,FS,TempFS) | List ],
                 HeadFS, FS) :-
   atomic( Atom ),
   apply_lex_unifs( T, Rules, List, HeadFS, TempFS ).

%  handle unifications of the form : Path1 = Path2
%   E.G.,
%    <head> = <head>
apply_lex_unifs( [ Features1=Features2 | T ] , Rules, List, HeadFS,
                 FS) :-
   find_path( main, Features1, Val, FS ),
   find_path( main, Features2, Val, FS ),
   apply_lex_unifs( T, Rules, List, HeadFS, FS ).



%-------------------------------------------------------
%
% find_path( PrevFeature, Features, Value, Var )
%
% Input :
%   PrevFeature - previous feature to get to this one
%   Features - list of features in this equation
%   Value - postion of feature in feature structure
%   Var - feature structure unifications are acting on
%
%
% Follow a path of features and return the value at the end.


%  do for each feature in list
find_path( Place, [ Head | Rest ] , Atom, Var ) :-
   % search in path information
   thepath( Place, Head, Var, Path ),
   find_path( Head, Rest, Atom, Path ).

%  stop at end of feature list
find_path( _Place, [], Atom, Atom ).



%-------------------------------------------------------
```

```
%
% find_type( Atom, Rule, Type )
%
% Input :
%   Atom - name of current template or lexical rule
%   Rule - top value on rule list
%
% Output :
%   Type - template or rule, depending on type of rule
%
%
% Get whether a lexical item is a template or lexical rule.


% lexical template
find_type( Atom, [ template(Atom,_Unifs) |  _Tail ] ,
           template ) :- !.

% lexical rule
find_type( Atom, [ lex_rule(Atom,_InFS,_OutFS,_Unifs) | _Tail ],
           rule ) :- !.

% keep searching
find_type( Atom, [ _Head | Tail ], Type ) :-
   find_type( Atom, Tail, Type ).
```

```
%  Module: COMPILEPATR.PL
%  Submodule: COMPILELEX.PL
%  Author: Susan B. Hirsh
%  Purpose: Compile all lexical entries.


% External predicates :
%
%
% Module PATRLIBRARY.PL
%
% write_clause/2 -
%    write clause to output stream in Prolog clause format.
%
%
% Module PATRSUPPORT.PL
%
% word/2 -
%    lexical entry from the database.


%-----------------------------------------------------------
%
% compile_lex( OutStream )
%
% Input :
%   OutStream - current output stream
%
%
% Precompile each lexical entry.  This involves actually
%  executing each one and then outputing the new entry as
%  lex( Word, FS ).


% execute the lexical entry and output it
compile_lex( OutStream ) :-
   word( Word, FS ),      % execute lexical entry
   % output to screen if trace flag is set
   ( trace_rules( yes ) ->
   format( 'LEXICAL ENTRY is ~w~n', [ (lex(Word,FS) ) ] )
   | true ),
   % write lexical entry to output file
   write_clause( ( lex(Word,FS) ), OutStream ),
```

```
   fail.                    % go to next lexical entry

% no more lexical entries
compile_lex( _OutStream ).
```

```
%  Module: PATRSUPPORT.PL
%  Author: Susan B. Hirsh
%  Purpose: Support module for the parser.


% Predicates necessary at runtime :
%
%
% LC Parser -
%
% patr/0 -
%    input loop using start symbol.
% parse/2 -
%    get all parses for a sentence
% print_parses/2 -
%    print parses and parse trees for a sentence.
% misc. LC predicates - parse/2, down/2, lc/4, and leaf/2
%
%
% Utility predicates -
%
% alphanumeric/1 -
%    character is alphanumeric.
% append/3-
%    concatenate two lists.
% case_shift/2 -
%    convert a list to lower case.
% concat/3 -
%    concatenate two atoms.
% digit/1 -
%    character is a digit.
% end_file/1 -
%    end of file character.
% find_category/2 -
%    find value of category attribute in a feature structure.
% format_stats/0 -
%    print runtime statistics.
% full_stop/1 -
%    '.'
% member/2 -
%    check membership in a list.
% new_line/1 -
%    new line character.
% string_size/2 -
```

```
%    get length of an atom.
% set_timer/0 -
%    set runtime timer.
% thepath/4 -
%     return the value of a path.
% upper/1 -
%     character is upper case.



% set up the system :


% declare type of predicates

:- dynamic(null/1).      % epsilon rules
:- dynamic(start/1).    % start symbol
:- dynamic(feature_order/3).    % path information

% operator definition
:- op(500,xfx,:).


% LC rules appear in two files
:- multifile lc/6.


:- no_style_check( all ).    % suppress warnings


:- ensure_loaded( pp ).       % load pretty-printer
:- ensure_loaded( readin ).  % load sentence reader



% predicates necessary for the LC parser to run


% initial calling sequence
parse( Cat, Tree ) --> down( Cat, Tree ).



% pick up a new left-corner once one has been processed
```

```
% epsilon rules
down( Cat, [] ) --> { null( Cat ) }.

% get next word and find whose left corner it is
down( Cat, Tree ) -->
   leaf( Child, OldTree ),
   lc( Child, Cat, OldTree, Tree ).



% every phrase is the left-corner of itself
lc( Type, Type, Tree, Tree ) --> [].



% this is a word
%  get the category information for parse tree
leaf( FS, Tree ) -->                % handle lexical entries
   [ Word ],                        % get the word
   { lex( Word, FS ) },             % get word's feature structure
   { find_category( FS, Cat ) },    % get category of feature structure
   { Tree =..[Cat,Word] }.          % make parse tree



%-------------------------------------------------------
%
% patr
%
%
% Read in a sentence and parse it with the start symbol.  By starting
%  the parse with a feature structure with the 'cat' feature
%  specified as the start symbol, the only good parses are those
%  that result in a parse whose 'cat' feature is that start symbol.


patr :-
   read_in( Sentence ),          % read in the sentence
   start( Symbol ),              % get the start symbol
   find_category( S, Symbol ),   % create filter for sentence parse
   parse( S, Sentence ).         % parse the sentence



%-------------------------------------------------------
```

```
%
% parse( Structure, Sentence )
%
% Input :
%   Structure - feature structure whose structure the parse must
%               match
%   Sentence - sentence to parse
%
%
% Get all parses for a sentence and print them out.  Read in a
%  new sentence and parse it.


% no more sentences
parse( S, end_of_file ):- !.

% parse sentences
parse( S, [] ) :- !,
   read_in( NewSentence ),
   parse( S, NewSentence ).

parse( S, Sentence ) :-
   set_timer,                    % set runtime timer
   % get all parses and parse trees for a sentence
   bagof( S-Tree, parse(S,Tree,Sentence,[]), Parses ),!,
   format_stats,                 % print runtime statistics
   print_parses( Parses,0 ),     % print parses
   read_in( NewSentence ),       % read in a new sentence
   parse( S, NewSentence ).      % parse that sentence

% error - couldn't parse
parse( S, NoParse ) :-
   format_stats,              % print runtime statistics
   % print error message
   format('~n*** Cannot parse ~w~n',[NoParse]),
   read_in( NewSentence ),    % read a new sentence
   parse( S, NewSentence ).   % parse that sentence



%-------------------------------------------------------
%
% print_parses( Parses, NumParses )
%
% Input :
```

```
%   Parses - list of parses and parse trees for a given sentence
%   NumParses - number of parses for the sentence
%
%
% Print the parses and parse trees for a sentence.


% print analysis on each parse
print_parses( [ Parse-Tree | Parses ], Count ) :-
   % increment count of number of parses
   NewCount is Count + 1,
   % print analysis
   format('~nAnalysis # ~d:~n~nParse Tree = ~w~n',[NewCount,Tree]),
   format('~p~n',[Parse]),
   print_parses( Parses, NewCount ).   % next parse

% no more parses
print_parses( [], Count ) :-
   % print count of number of parses
   format('~nNumber of Parses = ~d~n',[Count]).



%-------------------------------------------------------------
%
% concat( Atom1, Atom2, Atom )
%
% Input :
%   Atom1 - first atom
%   Atom2 - second atom
%
% Output :
%   Atom - concatenation of Atom1 and Atom2
%
%
% Concatenate two atoms.


concat( Atom1, Atom2, Result ) :-
   name( Atom1, List1 ),            % put in list form
   name( Atom2, List2 ),            % put in list form
   append( List1, List2, List3 ),   % concatenate as lists
   name( Result, List3 ).           % make into an atom
```

```
%-------------------------------------------------------
%
% append( List1, List2, List )
%
% Input :
%   List1 - first list
%   List2 - second list
%
% Output :
%   List - concatenation of List1 and List2
%
%
% Concatenate two lists.


% a list appended to the empty list is that list
append( [], List, List ).

% a list appended to another adds the head to a new list
append( [ Head | List1 ], List2, [ Head | List3 ] ) :-
   append( List1, List2, List3 ).



%-------------------------------------------------------
%
% find_category( FS, Cat )
%
% Input :
%   FS - a feature structure to get the category value from
%
% Output :
%   Cat - category value of the feature structure
%
%
% Get the value of the category attribute in the FS.

find_category( Lhs, Cat ) :-
   thepath( main, cat, Lhs, Cat ),   % get category of non-terminal
   atomic( Cat ), !.

% if Cat value isn't an atom, make it X
find_category( Lhs, x ).
```

```
%-----------------------------------------------------------
%
% member( Element, List )
%
% Input :
%   Element - element to check membership of
%   List - list to check membership in
%
%
% Check whether an element is a member of a list.

% element is head of the other list
member( Element, [ Element | _Rest ] ) :- !.

% keep searching the list
member( Element, [ _Head | Rest ] ) :-
   member( Element, Rest ).



%-----------------------------------------------------------
%
% set_timer
%
%
% Reset runtime timer.

set_timer :-
   statistics( runtime, [ _, _RunTime ] ).



%-----------------------------------------------------------
%
% format_stats
%
%
% Print runtime information


format_stats :-
   statistics( runtime, [ _, Stats ] ),    % get runtime
   Time is Stats/1000,                     % convert to seconds
```

```
   format( '~nRuntime = ~f~n', [ Time ] ). % print runtime.


%-------------------------------------------------------------
%
% string_size( Atom, Size )
%
% Input :
%   Atom - atom to get length of
%
% Output :
%   Size - number of characters in the atom
%
%
% Get the number of characters in an atom.


string_size( String, Size ) :-
   name( String, List ),   % make into a list
   length( List, Size ).   % get the length of the list



%-------------------------------------------------------------
%
% thepath( Node, Label, Term, Value )
%
% Input :
%   Node - start feature
%   Label - current feature
%
% Output :
%   Term - feature structure
%   Value - value of Label attribute
%
%
% Return of the value of the path <Node,Label>.


thepath( Node, Label, Term, Value ) :-
   % get the structure of the FS
   feature_order( Node, FS, Term ),
   member( Label:Value, FS ).         % get the value
```

```
%-------------------------------------------------------
%
% case_shift( Token, NewToken )
%
% Input :
%   Token - current input token
%
% Output :
%   NewToken - Token in all lower case.
%
%
% Convert token to all lower case.


%  if upper case, convert to lower
case_shift( [Upper | Mixed ], [ Letter | Lower ]) :-
   upper( Upper ), !,
   Letter is Upper+32,       % make lower case
   case_shift( Mixed, Lower ).

%  if not upper case, ignore
case_shift( [ Other | Mixed ], [ Other | Lower ]) :-
   case_shift( Mixed, Lower ).

%  no more to lower case
case_shift( [], [] ).



%-------------------------------------------------------
%
% alpha_numeric( Char )
%
% Input :
%   Char - character to check
%
%
% Check if a character is alphanumeric.


alpha_numeric(Ch) :-
   ( upper( Ch )           %  A..Z
   ; Ch >= 97, Ch =< 122   %  a..z
```

```
   ; digit( Ch )          %  0..9
   ; Ch = 95              %  '_'
   ; Ch = 63              %  '?'
   ; Ch = 42              %  '*'
   ; Ch = 39              %  non-standard "'"
   ; Ch = 96              %  non-standard "`"
   ).


%-----------------------------------------------------------
%
% digit( Char )
%
% Input :
%   Char - character to check
%
%
% Check if a character is a digit.


digit( Ch ) :-          %  0..9
   ( Ch >= 48
   , Ch =< 57
   ).



%-----------------------------------------------------------
%
% upper( Char )
%
% Input :
%   Char - character to check
%
%
% Check if a character is an upper case letter.


upper( Ch ) :-          %  A..Z
   ( Ch >= 65
   , Ch =< 90
   ).
```

```
% input delimiters

full_stop( 46 ).        %  '.'
end_file( -1 ).         % end of file
new_line( 10 ).         % new line
```

SRI International

CANDIDE, AN INTERACTIVE SYSTEM FOR THE ACQUISITION OF DOMAIN SPECIFIC KNOWLEDGE

Final Report
Covering the Period September 7, 1984 to January 31, 1988

May 1988

By: Fernando Pereira, Senior Computer Scientist

Prepared for:

Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, Virginia 22209

Attention: Commander Allen Sears

ARPA Order No. 3988

Contract N00039-84-C-0109

SRI Project 7783

Preparation of this paper was supported by the Department of the Navy under Contract N00039-84-C-0109 with the Space and Naval Warfare Systems Command.

Approved:

Ray Perrault, Director
Artificial Intelligence Center

Donald L. Nielson, Vice President and Director
Computer and Information Sciences Division

SRI International®